

O'REILLY®

6th Edition

Learning Python

Powerful Object-Oriented Programming



Mark Lutz

Learning Python

Get a comprehensive, in-depth introduction to the core Python language with this hands-on book. Based on author Mark Lutz's popular training course, this updated sixth edition will help you quickly write efficient, high-quality code with Python. It's an ideal way to begin, whether you're new to programming or a professional developer versed in other languages.

Complete with quizzes, exercises, and helpful illustrations, this easy-to-follow self-paced tutorial gets you started with Python 3.12 and all other releases in use today. With a pragmatic focus on what you need to know, it also introduces some advanced language features that have become increasingly common in Python code.

- Explore Python's built-in object types such as strings, lists, dictionaries, and files
- Create and process objects with Python statements, and learn Python's syntax model
- Use functions and functional programming to avoid redundancy and maximize reuse
- Organize code into larger components with modules and packages
- Code robust programs with Python's exception handling and development tools
- Apply object-oriented programming and classes to make code customizable
- Survey advanced Python tools including decorators, descriptors, and metaclasses
- Write idiomatic Python code that runs portably across a wide variety of platforms

Mark Lutz is the author of Python's classic and foundational texts, a former trainer with two decades of experience teaching Python to newcomers, and one of the people responsible for the prominence that Python enjoys today. For more author background, see Mark's books-and-software website at learning-python.com.

PYTHON / PROGRAMMING

US \$79.99 CAN \$99.99

ISBN: 978-1-098-17130-8



5 7 9 9 9

O'REILLY®

SIXTH EDITION

Learning Python

Powerful Object-Oriented Programming

Mark Lutz

O'REILLY®

Learning Python

by Mark Lutz

Copyright © 2025 Mark Lutz. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Louise Corrigan

Development Editor: Sara Hunter

Production Editor: Kristen Brown

Copyeditor: nSight, Inc.

Proofreader: Piper Content Partners

Indexer: nSight, Inc.

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

March 2025: Sixth Edition

Revision History for the Sixth Edition

2025-02-25: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098171308> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Learning Python*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-17130-8

[LSI]

To Vera.

You are my life.

Table of Contents

Preface.....	xxxi

Part I. Getting Started

1. A Python Q&A Session.....	3
Why Do People Use Python?	3
Software Quality	4
Developer Productivity	5
Is Python a “Scripting Language”?	5
OK, but What’s the Downside?	6
Who Uses Python Today?	7
What Can I Do with Python?	8
Systems Programming	8
GUIs and UIs	8
Internet and Web Scripting	8
Component Integration	9
Database Access	9
Rapid Prototyping	9
Numeric and Scientific Programming	10
And More: AI, Games, Images, QA, Excel, Apps...	10
What Are Python’s Technical Strengths?	10
It’s Object-Oriented and Functional	11
It’s Free and Open	11
It’s Portable	11
It’s Powerful	12
It’s Mixable	13
It’s Relatively Easy to Use	13
It’s Relatively Easy to Learn	13
Chapter Summary	14

Test Your Knowledge: Quiz	14
Test Your Knowledge: Answers	14
2. How Python Runs Programs.....	17
Introducing the Python Interpreter	17
Program Execution	18
The Programmer's View	18
Python's View	18
Execution-Model Variations	21
Python Implementation Alternatives	22
Standalone Executables	24
Future Possibilities	25
Chapter Summary	25
Test Your Knowledge: Quiz	25
Test Your Knowledge: Answers	26
3. How You Run Programs.....	27
Installing Python	27
Interactive Code	28
Starting an Interactive REPL	28
Where to Run: Code Folders	29
What Not to Type: Prompts and Comments	30
Other Python REPLs	30
Running Code Interactively	31
Why the Interactive Prompt?	32
Program Files	34
A First Script	34
Running Files with Command Lines	35
Command-Line Usage Variations	36
Other Ways to Run Files	37
Clicking and Tapping File Icons	37
The IDLE Graphical User Interface	38
Other IDEs for Python	39
Smartphone Apps	39
WebAssembly for Browsers	40
Jupyter Notebooks for Science	40
Ahead-of-Time Compilers for Speed	40
Running Code in Code	41
Other Launch Options	45
Which Option Should I Use?	46
Chapter Summary	46
Test Your Knowledge: Quiz	46
Test Your Knowledge: Answers	47
Test Your Knowledge: Part I Exercises	48

Part II. Objects and Operations

4. Introducing Python Objects.....	53
The Python Conceptual Hierarchy	53
Why Use Built-in Objects?	54
Python's Core Object Types	55
Numbers	56
Strings	57
Sequence Operations	58
Immutability	59
Type-Specific Methods	60
Getting Help	62
Other Ways to Code Strings	63
Unicode Strings	64
Lists	65
Sequence Operations	65
Type-Specific Operations	65
Bounds Checking	66
Nesting	66
Comprehensions	67
Dictionaries	69
Mapping Operations	69
Nesting Revisited	70
Missing Keys: if Tests	71
Item Iteration: for Loops	73
Tuples	74
Why Tuples?	75
Files	75
Unicode and Byte Files	76
Other File-Like Tools	77
Other Object Types	77
Sets	77
Booleans and None	78
Types	78
Type Hinting	79
User-Defined Objects	80
And Everything Else	80
Chapter Summary	81
Test Your Knowledge: Quiz	81
Test Your Knowledge: Answers	81
5. Numbers and Expressions.....	83
Numeric Object Basics	83
Numeric Literals	84

Built-in Numeric Tools	85
Python Expression Operators	85
Mixed Operators: Precedence	87
Parentheses Group Subexpressions	88
Mixed Types Are Converted Up	88
Preview: Operator Overloading and Polymorphism	89
Numbers in Action	89
Variables and Basic Expressions	89
Numeric Display Formats	91
Comparison Operators	92
Division Operators	94
Integer Precision	96
Complex Numbers	96
Hex, Octal, and Binary	96
Bitwise Operations	98
Underscore Separators in Numbers	99
Other Built-in Numeric Tools	101
Other Numeric Objects	103
Decimal Objects	103
Fraction Objects	104
Set Objects	106
Boolean Objects	112
Numeric Extensions	113
Chapter Summary	113
Test Your Knowledge: Quiz	113
Test Your Knowledge: Answers	114
6. The Dynamic Typing Interlude.....	115
The Case of the Missing Declaration Statements	115
Variables, Objects, and References	116
Types Live with Objects, Not Variables	118
Objects Are Garbage-Collected	118
Shared References	120
Shared References and In-Place Changes	121
Shared References and Equality	123
Dynamic Typing Is Everywhere	124
Type Hinting: Optional, Unused, and Why?	125
Chapter Summary	126
Test Your Knowledge: Quiz	126
Test Your Knowledge: Answers	127
7. String Fundamentals.....	129
String Object Basics	129
String Literals	131

Single and Double Quotes Are the Same	131
Escape Sequences Are Special Characters	132
Raw Strings Suppress Escapes	136
Triple Quotes and Multiline Strings	137
Strings in Action	138
Basic Operations	138
Indexing and Slicing	140
String Conversion Tools	143
“Changing” Strings Part 1: Sequence Operations	145
String Methods	146
Method Call Syntax	146
All String Methods (Today)	147
“Changing” Strings, Part 2: String Methods	148
More String Methods: Parsing Text	150
Other Common String Methods	151
String Formatting: The Triathlon	151
String-Formatting Options	152
The String-Formatting Expression	153
The String-Formatting Method	157
The F-String Formatting Literal	162
And the Winner Is...	167
General Type Categories	168
Types Share Operation Sets by Categories	168
Mutable Types Can Be Changed in Place	169
Chapter Summary	170
Test Your Knowledge: Quiz	170
Test Your Knowledge: Answers	170
8. Lists and Dictionaries.....	173
Lists	173
Lists in Action	175
Basic List Operations	175
Indexing and Slicing	176
Changing Lists in Place	177
More List Methods	180
Iteration, Comprehensions, and Unpacking	182
Other List Operations	184
Dictionaries	184
Dictionaries in Action	186
Basic Dictionary Operations	187
Changing Dictionaries in Place	187
More Dictionary Methods	188
Other Dictionary Makers	189
Dictionary Comprehensions	191

Key Insertion Ordering	192
Dictionary “Union” Operator	193
Intermission: Books Database	194
Dictionary Usage Tips	196
Chapter Summary	203
Test Your Knowledge: Quiz	203
Test Your Knowledge: Answers	203
9. Tuples, Files, and Everything Else.	207
Tuples	207
Tuples in Action	209
Why Lists and Tuples?	211
Records Revisited: Named Tuples	212
Files	213
Opening Files	214
Using Files	214
Files in Action	216
Text and Binary Files: The Short Story	217
Storing Objects with Conversions	218
Storing Objects with pickle	220
Storing Objects with JSON	221
Storing Objects with Other Tools	222
File Context Managers	223
Other File Tools	223
Core Types Review and Summary	224
Object Flexibility	225
References Versus Copies	226
Comparisons, Equality, and Truth	228
The Meaning of True and False in Python	231
Python’s Type Hierarchies	233
Type Objects	233
Other Types in Python	234
Built-in Type Gotchas	235
Assignment Creates References, Not Copies	235
Repetition Adds One Level Deep	236
Beware of Cyclic Data Structures	237
Immutable Types Can’t Be Changed in Place	237
Chapter Summary	237
Test Your Knowledge: Quiz	238
Test Your Knowledge: Answers	238
Test Your Knowledge: Part II Exercises	239

Part III. Statements and Syntax

10. Introducing Python Statements.....	245
The Python Conceptual Hierarchy Revisited	245
Python's Statements	246
A Tale of Two ifs	248
What Python Adds	248
What Python Removes	248
Why Indentation Syntax?	250
A Few Special Cases	252
A Quick Example: Interactive Loops	254
A Simple Interactive Loop	254
Doing Math on User Inputs	255
Handling Errors by Testing Inputs	256
Handling Errors with try Statements	257
Supporting Floating-Point Numbers	258
Nesting Code Three Levels Deep	259
Chapter Summary	259
Test Your Knowledge: Quiz	260
Test Your Knowledge: Answers	260
11. Assignments, Expressions, and Prints.....	261
Assignments	261
Assignment Syntax Forms	262
Basic Assignments	263
Sequence Assignments	264
Extended-Unpacking Assignments	266
Multiple-Target Assignments	271
Augmented Assignments	272
Named Assignment Expressions	274
Variable Name Rules	277
Expression Statements	280
Expression Statements and In-Place Changes	281
Print Operations	282
The print Function	282
Print Stream Redirection	285
Chapter Summary	288
Test Your Knowledge: Quiz	288
Test Your Knowledge: Answers	288
12. if and match Selections.....	291
if Statements	291
General Format	291
Basic Examples	292

Multiple-Choice Selections	293
match Statements	295
Basic match Usage	295
Advanced match Usage	297
Python Syntax Revisited	299
Block Delimiters: Indentation Rules	300
Statement Delimiters: Lines and Continuations	302
Special Syntax Cases in Action	303
Truth Values Revisited	304
The if/else Ternary Expression	306
Chapter Summary	307
Test Your Knowledge: Quiz	307
Test Your Knowledge: Answers	308
13. while and for Loops.	311
while Loops	311
General Format	311
Examples	312
break, continue, pass, and the Loop else	313
General Loop Format	313
pass	313
continue	314
break	315
Loop else	316
for Loops	318
General Format	318
Examples	318
Loop Coding Techniques	323
Counter Loops: range	324
Sequence Scans: while, range, and for	325
Sequence Shufflers: range and len	326
Skipping Items: range and Slices	327
Changing Lists: range and Comprehensions	327
Parallel Traversals: zip	328
Offsets and Items: enumerate	331
Chapter Summary	332
Test Your Knowledge: Quiz	332
Test Your Knowledge: Answers	332
14. Iterations and Comprehensions.	335
Iterations	336
The Iteration Protocol	337
Other Built-in Iterables	342
Comprehensions	347

List Comprehension Basics	347
List Comprehensions and Files	348
Extended List Comprehension Syntax	349
Comprehensions Cliff-Hanger	351
Iteration Tools	351
Other Iteration Topics	355
Chapter Summary	355
Test Your Knowledge: Quiz	356
Test Your Knowledge: Answers	356
15. The Documentation Interlude.....	357
Python Documentation Sources	357
# Comments	358
The dir Function	358
Docstrings and <code>__doc__</code>	360
Pydoc: The help Function	363
Pydoc: HTML Reports	366
Beyond Docstrings: Sphinx	370
The Standard Manuals	370
Web Resources	371
Common Coding Gotchas	372
Chapter Summary	373
Test Your Knowledge: Quiz	373
Test Your Knowledge: Answers	374
Test Your Knowledge: Part III Exercises	374

Part IV. Functions and Generators

16. Function Basics.....	379
Why Use Functions?	380
Function Coding Overview	381
Basic Function Tools	381
Advanced Function Tools	382
General Function Concepts	382
def Statements	383
return Statements	383
def Executes at Runtime	384
lambda Makes Anonymous Functions	384
A First Example: Definitions and Calls	385
Definition	385
Calls	386
Polymorphism in Python	386
A Second Example: Intersecting Sequences	387

Definition	387
Calls	388
Polymorphism Revisited	389
Segue: Local Variables	389
Chapter Summary	390
Test Your Knowledge: Quiz	390
Test Your Knowledge: Answers	390
17. Scopes.....	393
Python Scopes Basics	393
Scopes Overview	394
Name Resolution: The LEGB Rule	396
Scopes Examples	398
The Built-in Scope	399
The global Statement	401
Program Design: Minimize Global Variables	402
Program Design: Minimize Cross-File Changes	404
Other Ways to Access Globals	405
Nested Functions and Scopes	406
Nested Scopes Overview	406
Nested Scopes Examples	406
Closures and Factory Functions	407
Arbitrary Scope Nesting	409
The nonlocal Statement	409
nonlocal Basics	410
nonlocal in Action	410
nonlocal Boundary Cases	411
State-Retention Options	412
Nonlocals: Changeable, Per-Call, LEGB	413
Globals: Changeable but Shared	413
Function Attributes: Changeable, Per-Call, Explicit	414
Classes: Changeable, Per-Call, OOP	416
And the Winner Is...	416
Scopes and Argument Defaults	416
Loops Require Defaults, Not Scopes	418
Chapter Summary	419
Test Your Knowledge: Quiz	420
Test Your Knowledge: Answers	421
18. Arguments.....	423
Argument-Passing Basics	423
Arguments and Shared References	424
Avoiding Mutable Argument Changes	426
Simulating Output Parameters and Multiple Results	427

Special Argument-Matching Modes	428
Argument Matching Overview	428
Argument Matching Syntax	429
Argument Passing Details	430
Keyword and Default Examples	431
Arbitrary Arguments Examples	433
Keyword-Only Arguments	437
Positional-Only Arguments	439
Argument Ordering: The Gritty Details	440
Definition Ordering	440
Calls Ordering	442
Example: The min Wakeup Call	443
Full Credit	443
Bonus Points	445
The Punch Line	446
Example: Generalized Set Functions	446
Testing the Code	447
Example: Rolling Your Own Print	448
Using Keyword-Only Arguments	449
Chapter Summary	451
Test Your Knowledge: Quiz	451
Test Your Knowledge: Answers	452
19. Function Odds and Ends.....	455
Function Design Concepts	455
Recursive Functions	457
Summation with Recursion	457
Coding Alternatives	458
Loop Statements Versus Recursion	459
Handling Arbitrary Structures	460
Function Tools: Attributes, Annotations, Etc.	464
The First-Class Object Model	465
Function Introspection	466
Function Attributes	467
Function Annotations and Decorations	468
Anonymous Functions: lambda	470
lambda Basics	471
Why Use lambda?	472
How (Not) to Obfuscate Your Python Code	473
Scopes: lambdas Can Be Nested Too	475
Functional Programming Tools	475
Mapping Functions over Iterables: map	476
Selecting Items in Iterables: filter	477
Combining Items in Iterables: reduce	478

Chapter Summary	479
Test Your Knowledge: Quiz	479
Test Your Knowledge: Answers	479
20. Comprehensions and Generations.....	483
Comprehensions: The Final Act	483
List Comprehensions Review	484
Formal Comprehension Syntax	485
Example: List Comprehensions and Matrixes	487
Generator Functions and Expressions	490
Generator Functions: yield Versus return	490
Generator Expressions: Iterables Meet Comprehensions	496
Generator Functions Versus Generator Expressions	500
Generator Odds and Ends	501
Example: Shuffling Sequences	506
Scrambling Sequences	506
Permutating Sequences	509
Example: Emulating zip and map	513
Coding Your Own map	514
Coding Your Own zip and 2.X map	515
Asynchronous Functions: The Short Story	517
Async Basics	518
The Async Wrap-Up	524
Chapter Summary	525
Test Your Knowledge: Quiz	525
Test Your Knowledge: Answers	525
21. The Benchmarking Interlude.....	529
Benchmarking with Homegrown Tools	529
Timer Module: Take 1	530
Timer Module: Take 2	531
Timing Runner and Script	533
Iteration Results	535
More Module Mods	538
Benchmarking with Python's timeit	540
Basic timeit Usage	540
Automating timeit Benchmarking	544
Function Gotchas	549
Local Names Are Detected Staticallly	549
Defaults and Mutable Objects	550
Functions Without returns	552
Miscellaneous Function Gotchas	552
Chapter Summary	553
Test Your Knowledge: Quiz	553

Test Your Knowledge: Answers	553
Test Your Knowledge: Part IV Exercises	554

Part V. Modules and Packages

22. Modules: The Big Picture.....	559
Module Essentials	559
Why Use Modules?	560
Python Program Architecture	560
How to Structure a Program	561
Imports and Attributes	561
Standard-Library Modules	563
How Imports Work	563
Step 1: Find It	564
Step 2: Compile It (Maybe)	564
Step 3: Run It	567
The Module Search Path	567
Search-Path Components	567
Configuring the Search Path	570
The sys.path List	570
Module File Selection	571
Path Outliers: Standalones and Packages	572
Chapter Summary	573
Test Your Knowledge: Quiz	573
Test Your Knowledge: Answers	573
 23. Module Coding Basics.....	 575
Creating Modules	575
Module Filenames	575
Other Kinds of Modules	576
Using Modules	576
The import Statement	576
The from Statement	577
The from * Statement	577
Imports Happen Only Once	578
Imports Are Runtime Assignments	579
import and from Equivalence	580
Potential Pitfalls of the from Statement	580
Module Namespaces	582
How Files Generate Namespaces	582
Namespace Dictionaries: __dict__	583
Attribute Name Qualification	584
Imports Versus Scopes	585

Namespace Nesting	586
Reloading Modules	587
reload Basics	588
reload Example	588
reload Odds and Ends	590
Chapter Summary	590
Test Your Knowledge: Quiz	590
Test Your Knowledge: Answers	591
24. Module Packages.	593
Using Packages	593
Package Imports	594
Packages and the Module Search Path	594
Creating Packages	595
Basic Package Structure	595
Package <code>__init__.py</code> Files	597
Package <code>__main__.py</code> Files	599
Why Packages?	600
A Tale of Two Systems	600
The Roles of <code>__init__.py</code> Files	603
Package-Relative Imports	604
Relative and Absolute Imports	604
Relative-Import Rationales and Trade-Offs	605
Package-Relative Imports in Action	605
Namespace Packages	609
Python Import Models	609
Namespace-Package Rationales	610
The Module Search Algorithm	610
Namespace Packages in Action	611
Chapter Summary	613
Test Your Knowledge: Quiz	614
Test Your Knowledge: Answers	614
25. Module Odds and Ends.	617
Module Design Concepts	617
Data Hiding in Modules	618
Minimizing from <code>*</code> Damage: <code>__X</code> and <code>__all__</code>	619
Managing Attribute Access: <code>__getattr__</code> and <code>__dir__</code>	620
Enabling Language Changes: <code>__future__</code>	622
Dual-Usage Modes: <code>__name__</code> and <code>__main__</code>	622
Example: Unit Tests with <code>__name__</code>	623
The <code>as</code> Extension for <code>import</code> and <code>from</code>	625
Module Introspection	626
Example: Listing Modules with <code>__dict__</code>	627

Importing Modules by Name String	629
Running Code Strings	629
Direct Calls: Two Options	630
Example: Transitive Module Reloads	630
Module Gotchas	638
Module Name Clashes: Package and Package-Relative Imports	638
Statement Order Matters in Top-Level Code	638
from Copies Names but Doesn't Link	639
from * Can Obscure the Meaning of Variables	640
reload May Not Impact from Imports	640
reload, from, and Interactive Testing	641
Recursive from Imports May Not Work	642
Chapter Summary	643
Test Your Knowledge: Quiz	643
Test Your Knowledge: Answers	643
Test Your Knowledge: Part V Exercises	644

Part VI. Classes and OOP

26. OOP: The Big Picture.....	649
Why Use Classes?	649
OOP from 30,000 Feet	651
Attribute Inheritance Search	651
Classes and Instances	653
Method Calls	653
Coding Class Trees	654
Operator Overloading	656
OOP Is About Code Reuse	656
Chapter Summary	659
Test Your Knowledge: Quiz	659
Test Your Knowledge: Answers	659
27. Class Coding Basics.....	661
Classes Generate Multiple Instance Objects	661
Class Objects Provide Default Behavior	662
Instance Objects Are Concrete Items	662
A First Example	662
Classes Are Customized by Inheritance	664
A Second Example	665
Classes Are Attributes in Modules	666
Classes Can Intercept Python Operators	667
A Third Example	669
The World's Simplest Python Class	671

Classes: Under the Hood	671
Records Revisited: Classes Versus Dictionaries	673
Chapter Summary	675
Test Your Knowledge: Quiz	675
Test Your Knowledge: Answers	676
28. A More Realistic Example.....	679
Step 1: Making Instances	679
Coding Constructors	680
Testing as You Go	681
Using Code Two Ways	682
Step 2: Adding Behavior Methods	683
Coding Methods	684
Step 3: Operator Overloading	686
Providing Print Displays	686
Step 4: Customizing Behavior by Subclassing	688
Coding Subclasses	688
Augmenting Methods: The Bad Way	689
Augmenting Methods: The Good Way	689
Polymorphism in Action	691
Inherit, Customize, and Extend	692
OOP: The Big Idea	693
Step 5: Customizing Constructors, Too	693
OOP Is Simpler Than You May Think	695
Other Ways to Combine Classes: Composites	695
Step 6: Using Introspection Tools	698
Special Class Attributes	699
A Generic Display Tool	700
Instance Versus Class Attributes	701
Name Considerations in Tool Classes	702
Our Classes' Final Form	702
Step 7 (Final): Storing Objects in a Database	704
Pickles and Shelves	704
Storing Objects on a shelf Database	705
Exploring Shelves Interactively	706
Updating Objects on a Shelf	708
Future Directions	709
Chapter Summary	709
Test Your Knowledge: Quiz	710
Test Your Knowledge: Answers	710
29. Class Coding Details.....	713
The class Statement	713
General Syntax and Usage	713

Example: Class Attributes	714
Methods	716
Method Example	716
Other Method-Call Possibilities	717
Inheritance	718
Attribute Tree Construction	718
Inheritance Fine Print	719
Specializing Inherited Methods	719
Class Interface Techniques	721
Abstract Superclasses	722
Namespaces: The Conclusion	724
Simple Names: Global Unless Assigned	725
Attribute Names: Object Namespaces	725
The “Zen” of Namespaces: Assignments Classify Names	726
Nested Classes: The LEGB Scopes Rule Revisited	728
Namespace Dictionaries: Review	730
Namespace Links: A Tree Climber	732
Documentation Strings Revisited	734
Classes Versus Modules	735
Chapter Summary	736
Test Your Knowledge: Quiz	736
Test Your Knowledge: Answers	736
30. Operator Overloading.....	739
The Basics	739
Constructors and Expressions: <code>__init__</code> and <code>__sub__</code>	740
Common Operator-Overloading Methods	740
Indexing and Slicing: <code>__getitem__</code> and <code>__setitem__</code>	742
Intercepting Slices	743
Intercepting Item Assignments	744
But <code>__index__</code> Means As-Integer	745
Index Iteration: <code>__getitem__</code>	745
Iterable Objects: <code>__iter__</code> and <code>__next__</code>	746
User-Defined Iterables	747
Multiple Iterators on One Object	749
Coding Alternative: <code>__iter__</code> Plus <code>yield</code>	752
Membership: <code>__contains__</code> , <code>__iter__</code> , and <code>__getitem__</code>	756
Attribute Access: <code>__getattr__</code> and <code>__setattr__</code>	759
Attribute Reference	759
Attribute Assignment and Deletion	760
Other Attribute-Management Tools	761
Emulating Privacy for Instance Attributes: Part 1	761
String Representation: <code>__repr__</code> and <code>__str__</code>	762
Why Two Display Methods?	763

Display Usage Notes	764
Right-Side and In-Place Ops: <code>__radd__</code> and <code>__iadd__</code>	765
Right-Side Addition	766
In-Place Addition	770
Call Expressions: <code>__call__</code>	770
Function Interfaces and Callback-Based Code	772
Comparisons: <code>__lt__</code> , <code>__gt__</code> , and Others	773
Boolean Tests: <code>__bool__</code> and <code>__len__</code>	775
Object Destruction: <code>__del__</code>	776
Destructor Usage Notes	776
Chapter Summary	777
Test Your Knowledge: Quiz	777
Test Your Knowledge: Answers	778
31. Designing with Classes.....	779
Python and OOP	779
Polymorphism Means Interfaces, Not Call Signatures	780
OOP and Inheritance: “Is-a” Relationships	780
OOP and Composition: “Has-a” Relationships	782
Stream Processors Revisited	784
OOP and Delegation: “Like-a” Relationships	786
Pseudoprivate Class Attributes	788
Name Mangling Overview	789
Why Use Pseudoprivate Attributes?	789
Method Objects: Bound or Not	791
Bound Methods in Action	792
Classes Are Objects: Generic Object Factories	795
Why Factories?	796
Multiple Inheritance and the MRO	796
How Multiple Inheritance Works	797
How the MRO Works	799
Attribute Conflict Resolution	801
Example: “Mix-in” Attribute Listers	802
Example: Mapping Attributes to Inheritance Sources	811
Other Design-Related Topics	815
Chapter Summary	815
Test Your Knowledge: Quiz	815
Test Your Knowledge: Answers	816
32. Class Odds and Ends.....	817
Extending Built-in Object Types	817
Extending Types by Embedding	818
Extending Types by Subclassing	819
The Python Object Model	821

Classes Are Types Are Classes	821
Some Instances Are More Equal Than Others	822
The Inheritance Bifurcation	823
The Metaclass/Class Dichotomy	824
And One “object” to Rule Them All	825
Advanced Attribute Tools	826
Slots: Attribute Declarations	826
Properties: Attribute Accessors	834
__getattr__ and Descriptors: Attribute Implementations	836
Static and Class Methods	837
Why the Special Methods?	837
Plain-Function Methods	838
Static Method Alternatives	839
Using Static and Class Methods	840
Counting Instances with Static Methods	841
Counting Instances with Class Methods	842
Decorators and Metaclasses	845
Function Decorator Basics	845
A First Look at User-Defined Function Decorators	847
A First Look at Class Decorators and Metaclasses	848
For More Details	850
The super Function	851
The super Basics	851
The super Details	852
The super Wrap-Up	859
Class Gotchas	860
Changing Class Attributes Can Have Side Effects	860
Changing Mutable Class Attributes Can Have Side Effects, Too	861
Multiple Inheritance: Order Matters	862
Scopes in Methods and Classes	863
Miscellaneous Class Gotchas	864
“Overwrapping-itis”	865
Chapter Summary	865
Test Your Knowledge: Quiz	865
Test Your Knowledge: Answers	866
Test Your Knowledge: Part VI Exercises	866

Part VII. Exceptions

33. Exception Basics.....	875
Why Use Exceptions?	875
Exception Roles	876
Exceptions: The Short Story	877

Default Exception Handler	877
Catching Exceptions	878
Raising Exceptions	879
User-Defined Exceptions	880
Termination Actions	881
Chapter Summary	882
Test Your Knowledge: Quiz	883
Test Your Knowledge: Answers	883
34. Exception Coding Details.....	885
The try Statement	885
try Statement Clauses	885
The except and else Clauses	886
The finally Clause	892
Combined try Clauses	894
The raise Statement	898
Raising Exceptions	898
The except as hook	898
Scopes and except as	899
Propagating Exceptions with raise	900
Exception Chaining: raise from	900
The assert Statement	902
Example: Trapping Constraints (but Not Errors!)	903
The with Statement and Context Managers	904
Basic with Usage	904
The Context-Management Protocol	905
Multiple Context Managers	907
The Termination-Handlers Shoot-Out	908
Chapter Summary	909
Test Your Knowledge: Quiz	910
Test Your Knowledge: Answers	910
35. Exception Objects.....	911
Exception Classes	912
Coding Exceptions Classes	912
Why Exception Hierarchies?	914
Built-in Exception Classes	916
Built-in Exception Categories	917
Default Printing and State	918
Custom Print Displays	919
Custom State and Behavior	920
Providing Exception Details	921
Providing Exception Methods	921
Exception Groups: Yet Another Star!	923

Chapter Summary	925
Test Your Knowledge: Quiz	926
Test Your Knowledge: Answers	926
36. Exception Odds and Ends	927
Nesting Exception Handlers	927
Example: Control-Flow Nesting	928
Example: Syntactic Nesting	930
Exception Idioms	931
Breaking Out of Multiple Nested Loops: “go to”	931
Exceptions Aren’t Always Errors	932
Functions Can Signal Conditions with raise	933
Closing Files and Server Connections	934
Debugging with Outer try Statements	934
Running In-Process Tests	935
More on sys.exc_info	935
Displaying Errors and Tracebacks	937
Exception Design Tips and Gotchas	938
What Should Be Wrapped	938
Catching Too Much: Avoid Empty except and Exception	938
Catching Too Little: Use Class-Based Categories	940
Core Language Wrap-Up	941
The Python Toolset	941
Development Tools for Larger Projects	941
Chapter Summary	944
Test Your Knowledge: Quiz	945
Test Your Knowledge: Answers	945
Test Your Knowledge: Part VII Exercises	945

Part VIII. Advanced Topics

37. Unicode and Byte Strings	949
Unicode Foundations	950
Character Representations	950
Character Encodings	951
Introducing Python String Tools	954
The str Object	954
The bytes Object	954
The bytearray Object	955
Text and Binary Files	955
Using Text Strings	956
Literals and Basic Properties	956
String Type Conversions	957

Coding Unicode Strings in Python	959
Source-File Encoding Declarations	964
Using Byte Strings	966
Methods	966
Sequence Operations	967
Formatting	968
Other Ways to Make Bytes	968
Mixing String Types	969
The bytearray Object	970
Using Text and Binary Files	972
Text-File Basics	973
Text and Binary Modes	973
Unicode-Text Files	975
Unicode, Bytes, and Other String Tools	977
The re Pattern-Matching Module	977
The struct Binary-Data Module	978
The pickle and json Serialization Modules	978
Filenames in open and Other Filename Tools	980
The Unicode Twilight Zone	983
Dropping the BOM in Python	983
Unicode Normalization: Whither Standard?	987
Chapter Summary	989
Test Your Knowledge: Quiz	989
Test Your Knowledge: Answers	989

38. Managed Attributes.....	991
Why Manage Attributes?	991
Inserting Code to Run on Attribute Access	992
Properties	993
The Basics	993
A First Example	994
Computed Attributes	995
Coding Properties with Decorators	996
Descriptors	997
The Basics	998
A First Example	1000
Computed Attributes	1002
Using State Information in Descriptors	1003
How Properties and Descriptors Relate	1006
__getattr__ and __getattribute__	1007
The Basics	1008
A First Example	1010
Computed Attributes	1012
__getattr__ and __getattribute__ Compared	1014

Management Techniques Compared	1015
Intercepting Built-in Operation Attributes	1017
Example: Attribute Validations	1020
Using Properties to Validate	1021
Using Descriptors to Validate	1023
Using <code>__getattr__</code> to Validate	1027
Using <code>__getattribute__</code> to Validate	1028
Chapter Summary	1029
Test Your Knowledge: Quiz	1029
Test Your Knowledge: Answers	1030
39. Decorators.....	1031
What's a Decorator?	1031
Managing Calls and Instances	1032
Managing Functions and Classes	1032
Using and Defining Decorators	1032
Why Decorators?	1033
The Basics	1034
Function Decorator Basics	1034
Class Decorator Basics	1038
Decorator Nesting	1040
Decorator Arguments	1042
Decorators Manage Functions and Classes, Too	1042
Coding Function Decorators	1043
Tracing Function Calls	1043
Decorator State Retention Options	1044
Class Pitfall: Decorating Methods	1048
Timing Function Calls	1053
Adding Decorator Arguments	1055
Coding Class Decorators	1057
Singleton Classes	1057
Tracing Object Interfaces	1059
Class Pitfall: Retaining Multiple Instances	1062
Example: "Private" and "Public" Attributes	1064
Implementing Private Attributes	1064
Implementation Details I	1066
Generalizing for Public Declarations	1067
Implementation Details II	1070
Delegating Built-In Operations	1070
Example: Validating Function Arguments	1075
The Goal	1076
A Basic Range-Testing Decorator for Positional Arguments	1076
Generalizing for Keywords and Defaults	1078
Implementation Details	1082

Open Issues	1083
Decorator Arguments Versus Function Annotations	1085
Chapter Summary	1087
Test Your Knowledge: Quiz	1087
Test Your Knowledge: Answers	1088
40. Metaclasses and Inheritance.....	1089
To Metaclass or Not to Metaclass	1090
The Downside of “Helper” Functions	1090
Metaclasses Versus Class Decorators: Round 1	1092
The Metaclass Model	1093
Classes Are Instances of type	1093
Metaclasses Are Subclasses of type	1094
Class Statements Call a type	1095
Class Statements Can Choose a type	1096
Metaclass Method Protocol	1097
Coding Metaclasses	1097
A Basic Metaclass	1097
Customizing Construction and Initialization	1099
Other Metaclass Coding Techniques	1100
Managing Classes with Metaclasses and Decorators	1103
Inheritance: The Finale	1108
Metaclass Versus Superclass	1110
Metaclass Inheritance	1111
Python Inheritance Algorithm: The Simple Version	1112
Python Inheritance Algorithm: The Less Simple Version	1115
The Inheritance Wrap-Up	1117
Metaclass Methods	1118
Metaclass Methods Versus Class Methods	1119
Operator Overloading in Metaclass Methods	1120
Metaclass Methods Versus Instance Methods	1120
Chapter Summary	1122
Test Your Knowledge: Quiz	1122
Test Your Knowledge: Answers	1123
41. All Good Things.....	1125
The Python Tsunami	1125
The Python Sandbox	1127
The Python Upside	1127
Closing Thoughts	1128
Where to Go from Here	1128
Encore: Print Your Own Completion Certificate!	1128

Part IX. Appendixes

A. Platform Usage Tips..... 1135

B. Solutions to End-of-Part Exercises..... 1157

Index..... 1197

Preface

If you're browsing a bookstore and trying to make sense of this book, try this:

- *Python* is one of the most widely used programming languages in the world. It's part of nearly every role that computers play in our lives, and its relative ease of use makes it an ideal way to get started with programming.
- *This book* is a tutorial that teaches Python language fundamentals in depth. Its content is aimed at Python newcomers of all stripes, applies to every role that Python plays, and is based on decades of feedback from real learners like you.
- *This edition* updates this book for a decade of changes in Python and its world. It drops coverage of the now-defunct Python 2.X, explores new tools added to Python through version 3.12, and applies to other Pythons past and future.

The rest of this preface provides more background info on this book and its subject. It explains what's changed since the prior edition, debuts the book's examples package, and may help you get oriented before jumping into details.

Python

By most metrics you'll find on the web today, Python is now either the most-used programming language on the planet or very near the top of the list. The oft-cited TIOBE popularity index, for example, has ranked Python most popular for several years. As this edition is being written in 2024, it lists Python at #1 and well ahead of its nearest followers, C, C++, and Java.

Popularity ranks are prone to change, of course, and rely on usage metrics that are open to debate that we'll skip here. Based on every signal available, though, the Python revolution has clearly happened. It's now a ubiquitous, *go-to language* in web development, scientific programming, systems administration, AI, and nearly everything else computers do today. Thanks to its relative simplicity, Python is also commonly used to introduce newcomers to computer science across the education spectrum.

In fact, it's now fairly safe to say that Python played a pivotal role in *changing the world*. By spearheading a shift from statically typed compiled languages to dynamically typed scripting languages, Python ushered in changes that were at least as profound as those of the earlier transition from machine language to compiled languages. The scripting-language shift both enabled tasks formerly impractical, and opened the field to nonprofessional contributors. In the process, it propelled computers to a

prominence that would have been unthinkable in decades past. The internet, for example, simply could not be what it is today without tools like Python. For better and worse, Python enables the new.

Lofty goals aside, all of this has two tangible implications for this book. First, because this is now a post-revolution Python world, this edition does much less *cheerleading* than its predecessors. There's no reason to waste your time promoting a tool that's already arrived. This edition still summarizes Python's value proposition in **Chapter 1**, but the domains and tools that you're likely to explore after learning the basics here are readily available on the web and change too regularly to cover in a fundamentals book like this in any event.

Second, Python's popularity means that by reading this book, you'll be adding a *valuable skill* to your toolset, which will help you in a wide variety of computer-software tasks. Learning Python will both make entire domains accessible to you and enable you to achieve programming goals that might otherwise be difficult or impossible. Python users now enjoy a wealth of prior art ready to be leveraged in a language that accelerates their work.

That said, Python isn't the only game out there, and you'd also be well served by learning computer science from the ground up—the *full stack* in developer speak. Studying lower-level languages like C and Java, for example, can still give you a much more complete perspective than scripting languages alone and help you solve complex problems as they arise. Python itself, after all, is just a C program in its most-used flavor.

Even so, Python is a great place to start, and enough for many a task. While it's not without warts and suffers from the thrashing that's endemic to software today, a multitude of developers still find Python a lot more fun to use than other tools. Java and C++, for example, seem languages designed for middle management: they hobble programmers with training wheels and bureaucratic hurdles that have little to do with your program's goals. Python, in sharp contrast, remains *more ally than obstacle*. That viewpoint is naturally subjective, but you've come to the right place to do the math on this yourself.

This Book

This book is a tutorial on the Python language and a classic in its domain. It's the product of *three decades* spent using, promoting, and teaching Python, and dates back to the mid-1990s, when Python was still at version 1.X, and the web was just something developers mused about over lunch. Although the focus here is firmly on the present, that legacy naturally adds some historical context that will help you understand Python more deeply. Despite what you may have heard, the past matters, especially in knowledge-based fields.

Just as importantly, this book has always been based on live-and-in-person *feedback* from Python beginners struggling to learn Python for the first time. This feedback mostly owes to Python training classes taught over a period of two decades. While these classes have now gone the way of the dodo and Yahoo, this book takes care to retain its learner-inspired material because that's much—if not most—of its value.

As a result, if you're like most of the thousands of learners whose experiences have been captured here, you'll probably find that this book works like a self-paced version of the Python training sessions from which it arose. You may sometimes even find that it answers your questions before they are asked because a host of learners before you have had the same queries. This isn't clairvoyance; it simply reflects the fact that learning resources do best when they listen to learners.

It's also worth noting up front that this book sometimes *critiques* Python changes while presenting them. Critical thinking is crucial in engineering domains—especially in one caught up in an arms race that convolutes tools used by millions of people. On some levels, Python remains a constantly

morphing sandbox of ideas that too often prioritizes changer hubris over user need, and this book is not shy about calling this out. That said, the main goal here is to educate, not criticize, and opinions are always, well, opinionated. Although views here reflect decades of using and teaching Python, you should always judge the net worth of Python changes for yourself in whatever world you’ve been cast.

This Edition

This edition completely drops coverage of *Python 2.X*, the earlier version of the language, and adds new coverage of recent changes in *Python 3.X*, the newer and incompatible version. When the prior edition was published in 2013, Python 2.X was still widely used and probably even dominant. Because of that, the prior edition had to cover both the established 2.X and the new and upcoming 3.X, which at times made for a twisted tale indeed.

Over a decade later, 2.X has been officially sunsetted, and the Python world has adopted 3.X so fully that 2.X constitutes an unwarranted distraction to today’s Python learners. Hence, after a decades-long tenure, 2.X-specific content has been cut here to make room for new 3.X topics and address book size in general. Formally, this edition has been updated to be current with *Python 3.12* and its era, though it also previews 3.13 mods, and its focus on fundamentals makes it generally applicable to both older and newer Python versions.

Before the emails start flooding in, this book wants to make clear that it regrets the loss of historical context (and secretly pines for the simpler days of 2.X too). But 3.X is a substantial topic all by itself, without bifurcating the story and increasing the page count for a Python version that is now little used. So go well into that good Python night, 2.X, and long live 3.X. Unless stated otherwise, “Python” in this edition simply means the 3.X line in general and 3.12 and later in particular.

In terms of 3.X *mods*, this edition newly covers f‘...’ f-string literals, := named-assignment expressions, `match` statements, type hinting, `async` coroutines, star-unpacking proliferation, underscore digit separators, `__main__.py` package files, `__getattr__` module hooks, `except*` exception groups, dictionary-key insertion order, positional-only function arguments, hash-based bytecode files, and other additions, deprecations, and mutations that have cropped up over the last decade plus.

Among these, type hinting and `async` coroutines are not covered in depth—by design. The former is an optional and academic tool wholly unused by Python itself and at odds with its core principles. The latter is an advanced applications tool and has morphed constantly since its inception. And both quickly head over complexity cliffs that push them out of scope for Python beginners. When needed, supplemental info on such narrow topics is always just a search away on the web. Here, the goal is learning to walk well before trying to run.

Among other noteworthy changes this time around:

- The *Unicode* content in the advanced part’s [Chapter 37](#) is new and improved because this topic is now an essential in Python 3.X and the world at large.
- Usage coverage, including the new [Appendix A](#), gives more focus to *macOS*, *Android*, *Linux*, and *iOS* because not all of this book’s readers use Windows.
- Most code-file examples now have numbered *captions* because the extra formality distinguishes them better in the book, and it’s worth the space.
- Some *redundancy* has been trimmed, but not all, because repetition is useful and even important in learning resources.

- The *size* of this book was reduced by the prior bullet, rewrites and flow mods, and the net of 2.X cuts and 3.X inserts because it's less to grok.
- The size of the *print* version of this book was further reduced by moving two advanced but optional chapters online (Chapters 38 and 39) because it's less to lug.
- Fictitious *names* in examples are more gender neutral: “Bob” is now an ambiguous “Pat” unless paired with “Sue” as before because it better defuses bias.
- The *Monty Python* references have been dropped because they can be confusing and might be divisive, and borrowing personality from media seems cheap.
- Both *first-person voice* and *personal anecdotes* have been globally sacked because you've bought this book to learn Python, not an author's life story.

About the last two: Python's namesake was funny stuff, to be sure, but compulsively aping the work of a nearly all-male comedy group can seem like the secret handshake of an exclusive boys' club in hindsight. And while an occasional “I” or “my” might add color or credibility, overuse tends to come off as narcissism. Hence, the former 1k “spam” are now symbols more inclusive, and this book's three-decade tenure will have to speak for itself.

Despite all the mods, this edition remains much more *technical novel* than reference manual, and meaty enough to be comparable to a *full-semester class* on Python and programming. It introduces topics and expands them in later chapters as recurring themes, accumulating comprehensive coverage along the way. There are Python quick-reference resources at *python.org* and a multitude of blogs and videos that promise to teach Python rapidly. This book is for those who know that learning something well requires a bit more.

Media Choices

As of this writing, this book is destined to be available in three forms: *print* (i.e., paper), *ebook* (e.g., PDF, ePub, and Kindle), and *online* (a.k.a. web). The latter means the publisher's subscription service, currently branded as the O'Reilly learning platform (f.k.a. Safari). Naturally, each medium has valid uses that vary per reader. For instance, many prefer print for linear reads and electronic media for random searches and code copy/paste.

You're welcome to use the forms that work best for you, of course, but should carefully weigh the inherent *privacy trade-offs* of online media. By now, it should be abundantly clear that online anything comes with a potential for covert use and sale of customer information and access, which print and ebook media largely avoid. While monetization schemes vary, online users just might have a legion of salespeople peering over their shoulders as they use products for which they've already paid in full.

So please be careful out there. Unless you must use an employer's online subscription, this book suggests vetting your media options wisely and generally recommends its *print and ebook* forms to protect your privacy whenever possible. Your life really shouldn't be turned into a revenue stream unless you get reimbursed for it.

One last media tip: this book may also be fed by its publisher into *generative AI* models, the current hot topic in the tech press. Although this may prove useful for looking up isolated facts, it's not deep learning and isn't necessarily any more reliable than the least of the gossip it regurgitates. Until the world figures this out, please use wisely.

Updates and Examples

As most authors would attest, it's shockingly easy to miss typos in material you've read hundreds of times, and incompatible change is a norm in computer book topics. Hence, this edition, like all its predecessors, expects to be updated regularly after its publication.

This book's supplements, example files, and clarifications and corrections (a.k.a. errata) will all be maintained on the web. Here are the main coordinates for these online resources; as usual, consult your local search engine if these change over time:

Author website

This site will be used to post general *notes and updates* related to this text or Python itself—a hedge against future changes and a sort of virtual appendix to this book.

Book examples

This site will host this book's *examples package*, with both individual files to view and save online and a ZIP file of all the examples to download to your device.

Publisher website

This site will maintain this edition's *errata list*, and chronicle patches applied to the text in reprints. It will also link to other formats, including ebooks.

The second of these is home to the examples' *source code*. Please see the usage notes there, as well as the examples' coverage in “*Where to Run: Code Folders*” on [page 29](#). There is no plan to host the examples on *GitHub* today, because that site's learning curve is a lot to ask of beginners, and its commercial agendas should be cause for concern.

At any of the preceding websites, all *error reports* and *suggestions* for this book are welcome, and this feedback is invaluable for book quality. But please keep it fact-based and civil. Posts on the errata list have been mostly constructive, but the list has limited utility, and has been known to attract the usual trolls. Such is life in the age of global conversation.

Conventions and Reuse

This book's mechanics will make more sense once you start reading it, but as a reference, this book uses the following typographical conventions:

Italic

Used for email addresses, URLs, filenames, pathnames, and emphasizing new terms when they are first introduced

Constant width

Used for program code, the contents of files and the output from commands, and to designate modules, methods, statements, and system commands

Constant width bold

Used in code sections to show commands or text that would be typed by the user, and, occasionally, to highlight portions of code

Constant width italic

Used for replaceables (content you must fill in) and some comments in code sections



This element indicates a tip, suggestion, or general note relating to the nearby text. The icon may make more sense if you imagine a crow sounding an alarm.

Three more quick content notes here: first, you'll find occasional *sidebars* (delimited by boxes) and *foot-notes* throughout, which are often optional reading but provide additional context on the topics being presented. For instance, the sidebars that begin with “Why You Will Care:” amplify language topics with real-world use cases.

Second, Python *error messages* are often shortened in this book to conserve space. Please run offending code on your own device for the full text. Some messages include stack traces, and some have sprouted location indicators and speculative “Did you..?” help in the latest Python that might be useful for beginners, though veterans’ mileage may vary; either way, the extra text is excessive in a book tight on space.

Finally, the publisher maintains a standard statement about reusing code in this book, though the short, interactive code snippets used broadly here are hardly worth the legalese. Please see the book websites described earlier for the formal reuse story if you must care.

Acknowledgments

In keeping with the depersonalization goal discussed earlier, this edition will forego the usual lengthy acknowledgments of its predecessors. Instead, it extends simple gratitude to the scores of former students and readers, who largely shaped this book; the publishing company, which enabled this book to both reach learners and improve with time; the host of users, contributors, and promoters, who made Python what it is today; and the subject of this book’s dedication page, who patiently tolerated yet another book project. This one might be the last, but you never know what a bored author might next do.

PART I

Getting Started

A Python Q&A Session

If you're reading this book, you may already know what Python is and why it's an important tool to learn. If you don't, you probably won't be sold on Python until you've learned the language by reading the rest of this book and have done a project or two. But before we jump into details, this first chapter will briefly introduce some of the main reasons behind Python's popularity and begin sculpting a definition of the language. This takes the form of a question-and-answer session, which addresses some of the most common queries posed by beginners—like you.

Why Do People Use Python?

Because there are many programming languages to choose from, this is the usual first question of newcomers and a great place to start. Given that millions of people use Python today, there really is no way to answer this question with complete accuracy; the choice of development tools is often based on unique constraints or personal preference.

But after teaching Python to hundreds of groups and thousands of students, some common themes have emerged. The primary factors cited by Python users seem to be these:

Software quality

For many, Python's focus on readability, coherence, and software quality in general sets it apart from other tools in the programming world. Python code is designed to be *readable*, and hence reusable and maintainable—much more so than traditional scripting languages. The uniformity of Python code makes it relatively easy to understand, even if you did not write it. In addition, Python has deep support for more advanced *software reuse* mechanisms, such as object-oriented (OO) and functional programming, that can further promote code quality.

Developer productivity

Python boosts developer productivity many times beyond compiled or statically typed languages. As one measure of this, Python code is typically *one-third to one-fifth* the size of equivalent C++ or Java code. That means there is less to type, less to debug, and less to maintain after the fact. Most Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed.

Program portability

Python programs generally run unchanged on *all major computer platforms*. Porting a Python program between Linux and Windows, for example, is often just a matter of copying its code between machines. Moreover, Python offers multiple options for coding portable graphical user interfaces, database access programs, web-based systems, and more. Even operating system interfaces as proprietary as program launches and directory processing are as portable in Python as they can possibly be.

Application support

Python comes with a large collection of prebuilt and portable functionality, known as the *standard library*. This library supports an array of application-level programming tasks, from text pattern matching to network scripting. In addition, Python can be extended with both homegrown libraries and a vast collection of third-party software. As covered ahead, Python's *third-party domain* offers tools for website construction, numeric programming, AI, and much more. The NumPy extension, for instance, has elevated Python to a core tool in science, technology, engineering, and math (STEM), and Django and PyTorch have done similar for the web and AI.

Component integration

Python scripts can communicate with other parts of an application using a variety of integration mechanisms. Such integrations allow Python to be used as a product *customization and extension* tool. For instance, Python code can invoke compiled libraries and be run by compiled programs, interact with other components over networks, and use Android and iOS toolkits on smartphones. In fact, many Python core tools, including files, ultimately use precoded interfaces to system libraries; even if your program is all Python, it's not standalone.

Love of craft

Because of Python's ease of use and built-in toolset, it can make the act of programming *more pleasure than chore*. Although this benefit is intangible and subjective, its effect on productivity is an important asset. People do get paid for coding Python, but many use it just for *fun*—a testimonial rare in the software field.

Of these factors, the first two—quality and productivity—are probably the most compelling benefits to most Python users; let's take a closer look at each.

Software Quality

By design, Python has a deliberately simple and readable syntax and a highly consistent programming model. As a slogan at an early Python conference attested, the net result is that Python seems to *fit your brain*—that is, features of the language interact in consistent and limited ways and follow naturally from a small set of core concepts. This makes the language easier to learn, understand, and remember. In practice, Python programmers do not need to constantly refer to manuals when reading or writing code; it generally just makes sense.

By philosophy, Python adopts a somewhat minimalist mindset. This means that although there are usually multiple ways to accomplish a coding task, there is usually just one obvious way, a few less obvious alternatives, and a small set of coherent interactions throughout. Moreover, Python usually doesn't make arbitrary decisions for you; when interactions are ambiguous, explicit intervention is preferred over “magic.” In the Python way of thinking, explicit is better than implicit, and simple is better than complex.

Beyond such design themes, Python includes tools such as modules and object-oriented programming (OOP) that naturally promote code reusability in skilled hands of the sort you're about to acquire. And because Python is focused on quality, most Python programmers naturally are too; this can be a crucial advantage when it's time to use someone else's Python code.

Developer Productivity

If you've worked in the software field, you know that it can be a dynamic and bumpy ride. During the great internet boom of the mid-to-late 1990s, it was difficult to find enough programmers to implement software projects; developers were asked to implement systems as fast as the internet evolved. In later eras of economic recession and layoffs, the picture shifted; programming staffs were often asked to accomplish the same tasks with even fewer people.

In both of these scenarios, Python has shined as a tool that allows programmers to get more done with less effort. It is explicitly optimized for *speed of development*—its simple syntax, dynamic typing, lack of compile steps, and built-in toolset allow programmers to develop programs in a fraction of the time needed when using some other tools.

The net effect is that Python typically increases developer productivity many times beyond the levels supported by traditional languages and often makes the impossible possible. That's good news in both boom and bust times, and everywhere the software industry goes in between.

Is Python a “Scripting Language”?

Maybe. Python is often applied in scripting roles, but not always. It's regularly called an *object-oriented scripting language*—a definition that blends support for OOP with an orientation toward scripting contexts, but this may be too narrow and dismissive. If pressed for a one-liner, Python is probably better known as:

A general-purpose programming language that blends procedural, functional, and object-oriented paradigms and accelerates software development by reducing complexity.

That may not fit on a t-shirt quite as well, but it captures both the richness and scope of today's Python.

Nevertheless, the term *scripting* seems to have stuck to Python like glue, perhaps as a contrast with the larger efforts required by some other tools. For example, some people use the word “script” instead of “program” to describe a Python code file, because it seems simpler and less formal to code. More usefully, others reserve “script” for a top-level file, and “program” for a more sophisticated multifile application, both of which are common in Python.

Because the term *scripting language* has so many different meanings to different observers, though, some would prefer that it not be applied to Python at all. In fact, people tend to make three very different assumptions when they hear Python labeled as such, some of which are more useful than others:

Shell tools

Sometimes when people hear Python described as a scripting language, they think it means that Python is a tool for coding operating-system-oriented scripts. Such programs are often launched from console command lines and perform tasks such as processing text files and launching other programs.

As you'll learn ahead, Python programs can and do serve such roles, but this is just one of dozens of common Python application domains. It is not just a better shell-script language.

Control language

To others, scripting refers to a “glue” layer used to control and direct (i.e., script) other application components. As noted earlier, Python programs are indeed often deployed in the context of larger applications. For instance, to test hardware devices, Python programs may call out to components that give low-level access to a device or port. Similarly, programs may run bits of Python code at strategic points to support end-user product customization without the need to ship and recompile the entire system’s source code.

Python’s simplicity makes it a naturally flexible control tool. Technically, though, this is also just a common Python role; many (and perhaps most) Python programmers code standalone scripts without ever using or knowing about any integrated components. It is not just a control language.

Ease of use

Probably the best use of the term *scripting language* is to refer to a relatively simple language used for coding tasks quickly. This is especially true when the term is applied to Python, which allows much faster program development than compiled languages like C++ or larger languages like Java. Its rapid development cycle fosters an exploratory, incremental mode of programming that has to be experienced to be appreciated.

Don’t be fooled, though—Python is not just for simple tasks. Rather, it makes tasks simple by its ease of use and flexibility. Python has an approachable feature set, but it allows programs to scale up in sophistication as needed. Because of that, it is commonly used for both quick tactical tasks and longer-term strategic development.

So, is Python a scripting language or not? It depends on whom you ask (and perhaps when you ask them). In general, the term *scripting* is best reserved for the rapid and flexible mode of development that Python supports, rather than a particular application domain or limiting label.

On a related note, people also sometimes call Python an *interpreted language* to distinguish it from languages like C and C++. While this is also sometimes meant to pigeonhole, it’s also easier to dismiss: there are many implementations of Python today, spanning the spectrum from traditional interpreters to traditional compilers, so “interpreted” doesn’t apply. The clearer distinction may be that Python is *dynamically typed*, not statically typed like languages that are normally compiled. As you’ll soon learn, this accounts for much of the power that Python brings to development tasks.

OK, but What’s the Downside?

The only universally recognized downside to Python that has emerged over its more than three-decade tenure may also be an inevitable trade-off for its ease of use: Python’s *execution speed* may not always be as fast as that of fully compiled and lower-level languages such as C and C++. Though relatively rare today, you may still occasionally need to get “closer to the iron” for some tasks by using languages that are more directly mapped to the underlying hardware.

We’ll talk about implementation concepts in [Chapter 2](#), but in short, the most-used versions of Python today compile (i.e., translate) source code statements to an intermediate format known as *bytecode* and then interpret the bytecode. Bytecode provides portability, as it is a platform-independent format. However, because Python is not commonly compiled all the way down to binary *machine code* (e.g., instructions for an Intel or ARM chip in your PC or phone), some programs will run more slowly in Python than in a fully compiled language like C.

Whether you will ever *care* about this execution speed difference depends on what kinds of programs you write. Python is regularly optimized, and Python code runs fast enough by itself in most application domains. Furthermore, whenever you do something “real” in a Python script, like processing a file

or constructing a graphical user interface (*GUI*), your program will actually run at C speed, because such tasks are immediately dispatched to compiled C code inside the Python interpreter.

And really, Python's speed is a bit of a red herring today: in truth, Python *is* commonly used in domains that require optimal execution speeds. Numeric programming and computer games, for example, often need at least their core number-crunching components to run at C speed (or better), but are frequently coded in Python nevertheless.

There are multiple ways to achieve speed in Python when it counts. For instance, systems such as *PyPy* compile bytecode further as your program runs; *Cython* allows you to code in a C-and-Python hybrid that's compiled in full; and crucial code can be split off to *compiled extensions* linked into Python for use in scripts. As an example, the *NumPy* numeric-programming extension combines compiled and optimized libraries with the Python language, and in the process turns Python into a numeric programming tool that is simultaneously efficient and easy to use. Indeed, NumPy Python code regularly achieves speed parity with Fortran and C++, without their added complexities.

More fundamentally, though, execution speed is not the only priority in most software development. Python's *speed-of-development* gain is often far more important than any speed-of-execution loss, especially given modern computer speeds—and modern computer deadlines. Naturally, Python has other *potential* downsides, including its frequent changes and convolutions, but these are subjective calls best made after you've had a chance to vet them in this book.

Who Uses Python Today?

Because Python is a free and open source software (*FOSS*) tool, an accurate count of its user base is impossible—there are no license registrations to tally. Moreover, Python has been automatically included with countless Linux distributions, Macintosh computers, and a wide range of products and hardware, further clouding the user-base picture.

In general, though, Python enjoys a very large user base and an active developer community. It is generally considered to be among the *top 5* most widely used programming languages in the world today (and for true sports fans, often weighs in at #1). Because Python has been broadly used for *over three decades*, it's also very stable and robust.

Because of all this, Python is regularly applied in real revenue-generating products by real *companies*. While user lists are prone to change, a list of notable and known companies using Python today reads like a who's who of the software field: Google, Intel, Disney, YouTube, Industrial Light & Magic, Red Hat, NASA, Eve Online, Seagate, JPL, Hewlett-Packard, JP Morgan Chase, Dropbox, ESRI, Instagram, Spotify, Pinterest, Reddit, Microsoft, Netflix, and many more.

More broadly, Python is deployed by most organizations developing software today in either strategic or tactical roles and regularly serves as the tool of choice in computer science education. It's not just for one thing, it's for everything.

For a sampling of additional Python users and applications that's hopefully more up to date than a book can ever be, try the following pages at Python's site: [Python Success Stories](#), [Applications for Python](#), and [Quotes about Python](#). As usual, you may also be able to uncover other Python roles of interest with a web search in your local browser or app.

What Can I Do with Python?

Commercial applications may be compelling, but people also use Python for all sorts of real-world, day-in/day-out tasks, whether for profit, need, hobby, or fun. In fact, as a general-purpose language, Python's roles are virtually unlimited: you can use it for everything from gaming and website development to robotics and content management.

That said, Python roles seem to fall into a few broad categories. The next few sections survey some of Python's most common application domains today, as well as prominent tools used in each domain.

Two notes up front: first, we won't be able to explore these tools in any depth either here or in this book at large. NumPy and Django, for example, are book-length topics on their own, and our goal in this book is to learn the *Python* code you will be writing to use systems like these. Second, apologies in advance to the many other tools omitted here only for space; if you are interested in any of the following topics, please search online for more tools and resources.

Systems Programming

Python's built-in interfaces to operating-system services make it ideal for writing portable and maintainable system-administration utilities—sometimes called *shell* or *command-line* tools, though they may be used in numerous ways. Python programs can search files and directory trees, launch and configure other programs, do parallel processing with processes, threads, and coroutines, and more.

Python's standard library comes with Portable Operating System Interface (*POSIX*) bindings and support for all the usual OS tools, including environment variables, files, sockets, pipes, processes, threads, regular-expression pattern matching, command-line arguments, standard-stream interfaces, shell-command launchers, filename expansion, ZIP file utilities, and XML, JSON, and CSV parsers. In addition, the bulk of Python's system interfaces are designed to be portable; for example, a script that copies directory trees typically runs unchanged on all platforms that host Python.

GUIs and UIs

Python's simplicity and rapid turnaround also make it a good match for GUI programming on devices of all kinds. For instance, Python comes with a standard object-oriented interface to the Tk GUI toolkit called *Tkinter* (and *tkinter* in code), which allows Python programs to implement portable GUIs with a native look and feel. Python/Tkinter GUIs run unchanged on Windows, macOS, and Linux, and on Android with the help of a freeware app today (see [Appendix A](#)).

In addition, third-party tools offer other routes to portable UIs—including both traditional GUIs like *Kivy*, *BeeWare's Toga*, *PyQt*, and *wxPython*; and web-browser based solutions like *Django*, *Flask*, and *WebAssembly*. If your app, like most, must interact with users, Python has multiple ways to write once and run everywhere.

Internet and Web Scripting

Python comes with standard internet modules that allow programs to perform a wide variety of networking tasks, in both client and server modes. Scripts can communicate over sockets; extract form information sent to server-side CGI web scripts; transfer files by FTP; parse and generate XML and JSON documents; compose and send, and receive and parse email; fetch web pages by URLs and parse their HTML; and more.

In addition, a large collection of third-party tools are available on the web for doing internet programming in Python. For example, web-development frameworks, such as *Django*, *Flask*, *TurboGears*, and *Zope*, support construction of full-featured and production-quality websites with Python. Many of these include tools like object-relational mappers, server-side scripting and templating, and AJAX support, to provide complete and enterprise-level web development solutions. The *Pyjamas* Python-to-JavaScript transpiler; the *Beautiful Soup* HTML content extractor; and the *WebAssembly*, *Pyodide*, *py2wasm*, and *PyScript* Python-in-the-browser enablers provide even more web possibilities.

Component Integration

We discussed the component integration role earlier. Python's ability to be extended by and embedded in systems coded in C, C++, and Java makes it useful as a flexible glue language for scripting the behavior of other software components. For instance, integrating a C library into Python allows Python to test and launch the library's tools, and embedding Python in a product enables on-site customizations to be coded without having to recompile the entire product (or ship its source code at all).

Tools such as *SWIG*, *Boost.Python*, *CFFI*, and *HPy* automate much of the work needed to link compiled components with Python scripts, and the *Cython* system allows coders to mix Python and C-like code to create compiled extensions. Other tools provide more ways to script components—including the *pyjnius* and *Chaquopy* Python/Java bridges; *pywin32*'s support for Windows Component Object Model (COM); the *REST*, *SOAP*, and *XML-RPC* cross-network conduits; and the *Jython* Java and *IronPython* .NET implementations of Python. In short, most software components are in scope to Python code.

Database Access

For traditional database demands, there are Python interfaces to all commonly used relational database systems—Oracle, MySQL, PostgreSQL, Informix, ODBC, SQLite, and more. The Python world has also defined a *portable database API* for accessing SQL database systems from Python scripts, which looks the same on a variety of underlying database implementations. For basic program-storage needs, Python comes with built-in support for its own *pickle* objects, language-agnostic *JSON* documents, and the in-process *SQLite* embedded SQL database engine.

Also for Python scripts, *PyYAML* parses and emits *YAML* data; *ZODB* and *Durus* provide object-oriented databases; *SQLObject* and *SQLAlchemy* implement object relational mappers (ORMs), which graft Python classes onto relational tables; and *PyMongo* interfaces to MongoDB, a high-performance *JSON*-style document database. Python can also access cloud storage options in Google's *App Engine*, Microsoft's *Azure*, and Amazon's *AWS*.

Rapid Prototyping

To Python programs, components written in Python and C look the same. Because of this, it's possible to prototype systems in Python initially, and then move selected components to a fully compiled language such as C or C++ for delivery. Because Python doesn't require a complete rewrite once the prototype has solidified, the parts of the system that don't need the efficiency of a compiled language can remain coded in Python for ease of maintenance and use. But beware: given the many optimization routes you met earlier, your prototype may very well be your deliverable.

Numeric and Scientific Programming

Python is also widely used in numeric programming—a domain that would not traditionally have been considered to be in the scope of scripting languages but has grown to become one of Python’s most compelling use cases. Prominent here, the *NumPy* high-performance numeric-programming extension for Python mentioned earlier includes such advanced tools as an array object, interfaces to standard mathematical libraries, and much more. By integrating Python with numeric routines coded in a compiled language for speed, NumPy turns Python into a sophisticated yet easy-to-use numeric-programming tool that can often replace code written in traditional languages like Fortran or C++.

Additional numeric tools often use NumPy as a core component, and add support for visualization, parallel processing, statistical analysis, and more. For example, *SciPy* provides libraries of scientific programming tools; *pandas* supports data analysis; *matplotlib* adds visualization tools; *Jupyter* notebooks are geared toward math workers’ needs; *Numba* and *PyThran* offer just-in-time (JIT) and ahead-of-time (AOT) compilers for Python numeric code, respectively; and the *Cython* and *PyPy* systems noted earlier can optimize algorithmic code. For more basic needs, Python itself comes with a statistics module; complex, fixed-point, and rational math; and other tools you’ll learn about in this book.

And More: AI, Games, Images, QA, Excel, Apps...

Python is commonly applied in far more domains and with far more tools than can possibly be covered here. For example, it’s also used in:

- Artificial intelligence (see *PyTorch*, *TensorFlow*, and *Keras*)
- Game programming (see *pygame*, *Panda3D*, and *Kivy*)
- Image and graphics processing (see *Pillow*, *PyOpenGL*, and *OpenCV*)
- Quality assurance and testing (see *PyTest*, *unittest*, and *Selenium*)
- Excel spreadsheets (see *xlwings*, *PyXLL*, and *Excel*)
- Smartphone apps (see *Kivy*, *BeeWare*, and [Appendix A](#))
- Microcontrollers and ports (see *MicroPython* and *PySerial*)
- And of course, much more

For links to resources in these and many other fields, try Wikipedia’s Python [software page](#); the [PyPI website](#), which hosts extension packages installed by Python’s *pip*; and the normal web searches.

Though application domains underscore Python’s practical utility, keep in mind that many are largely just instances of Python’s component *integration* role in action again. Adding Python as a frontend to libraries of components written in a compiled language like C makes Python useful for scripting in a wide variety of roles. As a general-purpose language that supports integration, Python is broadly, if not universally, applicable.

What Are Python’s Technical Strengths?

Naturally, this is a developer’s question. If you don’t already have a programming background, the terminology in the next few sections may be a bit baffling—don’t worry, we’ll explore all of these topics in more detail as we proceed through this book. For both current and future developers, though, here is a quick rundown of Python’s top technical features. Some have been touched on earlier, but this section fills in more of the story.

It's Object-Oriented and Functional

Python is an object-oriented language, from the ground up. As you'll find in this book, its *class model* supports advanced notions such as polymorphism, operator overloading, and multiple inheritance; yet, in the context of Python's simple syntax and typing, OOP is remarkably easy to apply. In fact, if you don't understand these terms, you'll find they are much easier to learn with Python than with just about any other OOP language available.

Of equal significance, OOP is an *option* in Python; you can go far without having to become an object guru all at once. Much like C++, Python supports both procedural and object-oriented programming modes. Its object-oriented tools can be applied if and when constraints allow. This is especially useful in tactical development modes, which often preclude the design phases that best utilize OOP's benefits.

In addition to its original *procedural* (statement-based) and *object-oriented* (class-based) paradigms, Python today has built-in support for *functional* programming—a set that by most measures includes generators, comprehensions, closures, maps, decorators, anonymous-function lambdas, and first-class function objects. As you'll also learn in this book, these can serve as both complement and alternative to its OOP tools.

It's Free and Open

Python is completely free to use and distribute. As with other *open source software*, such as Linux and Apache, you can fetch the entire Python system's source code for free on the internet. There are no restrictions on copying it, embedding it in your systems, or shipping it with your products.

But don't get the wrong idea: “free” doesn't mean “unsupported.” On the contrary, Python users have access to numerous online resources that respond to queries and issues quicker than most commercial software. Moreover, because Python comes with complete source code, it empowers developers in ways that closed commercial software cannot. Although studying or changing a programming language's implementation code isn't everyone's idea of fun, it's comforting to know that you can. You're not dependent on the whims of a commercial vendor, because the ultimate documentation—*source code*—is at your disposal as a last resort.

It's Portable

We touched on portability earlier. The standard implementation of Python is written in portable ANSI C, and compiles and runs on virtually every major platform currently in use. For example, Python programs run today on everything from smartphones to supercomputers. As a partial list, Python is available on Windows and macOS PCs, Linux and Unix workstations and servers, Android and iOS smartphones and tablets, real-time systems like VxWorks, Cray supercomputers, IBM mainframes, and more. Moreover, this list expands regularly; Android and iOS, for example, are gaining official *python.org* support at this writing.

Like the language itself, the *standard-library* modules that ship with Python are designed to be portable across platform boundaries; their file tools, for instance, remove many or most of the proprietary aspects of storage on some hosts. Furthermore, Python programs are automatically compiled to portable *bytecode*, which runs the same on any platform with a compatible version of Python installed (more on this in the next chapter), and there are multiple ways to code portable *user interfaces* in Python with traditional GUIs and web-based options described earlier.

This means that programs that use the Python language, its standard libraries, and portable extensions run the same on most systems that host a Python interpreter. Python also supports platform-specific

extensions (e.g., *pywin32* on Windows, *PyObjC* on macOS, and *pyjnius* on Android), but Python itself works the same everywhere.

It's Powerful

From a features perspective, Python is something of a hybrid. Its toolset places it between traditional scripting languages (such as Tcl, Scheme, and Perl) and systems development languages (such as C, C++, and Java). Python provides all the simplicity and ease of use of a scripting language, along with more advanced software-engineering tools typically found in compiled languages. Unlike some scripting languages, this combination makes Python useful for large-scale development projects. As a preview, here are some of the main things you'll find in Python's toolbox:

Dynamic typing

Python keeps track of the kinds of objects your program uses when it runs, and doesn't require complicated type and size declarations in your code. In fact, as you'll see in [Chapter 6](#), there is no such thing as a type or variable declaration anywhere in Python (apart from recent "hinting," which Python itself does not use). Because Python code does not constrain data types, it is also usually automatically applicable to a whole range of objects.

Automatic memory management

Python automatically allocates objects and reclaims ("garbage collects") them when they are no longer used. As you'll learn, Python keeps track of objects in use and the memory they hold, so you don't have to.

Programming-in-the-large support

For building larger systems, Python includes tools such as modules, classes, and exceptions. These tools allow you to organize systems into components, use OOP to reuse and customize code, and handle events and errors gracefully. Python's functional programming tools, described earlier, meet some of the same goals.

Built-in object types

Python provides commonly used data structures such as lists, dictionaries, and strings as intrinsic parts of the language. As you'll see, its built-in objects are both flexible and easy to use. They can grow and shrink on demand, can be arbitrarily nested to represent complex information, and are immune to common memory errors.

Built-in tools

To process all those object types, Python comes with powerful and standard operations, including concatenation (joining collections), slicing (extracting sections), sorting, mapping, and more.

Library utilities

For more specific tasks, Python also comes with a large collection of precoded library tools that support everything from regular expression pattern matching to network servers. Once you learn the language itself, Python's library tools are where much of the application-level action occurs.

Third-party utilities

Because Python is open source, developers are encouraged to contribute precoded tools that support tasks beyond those supported by its built-ins. On the Web, you'll find free support for image processing, numeric programming, database access, website development, formal testing, and much more (see ["What Can I Do with Python?" on page 8](#)).

Despite the array of tools in Python, it retains a noticeably simple syntax and design. The result is a powerful programming tool with all the usability of a scripting language.

It's Mixable

As noted earlier, Python programs can easily be “glued” to components written in other languages in a variety of ways—both locally and across networks. That means you can add functionality to the Python system as needed and use Python programs within other environments or systems.

Mixing Python into systems coded in more demanding languages, for instance, makes it an easy-to-use frontend language for testing and customization. As also mentioned earlier, this makes Python good at rapid prototyping too—systems may be coded in Python first for development speed and later moved to extensions one piece at a time for execution speed if and when needed.

It's Relatively Easy to Use

Compared to alternatives like C++, Java, and C#, Python programming seems astonishingly simple to most observers. To run Python code in most contexts, you simply type it and run it. There are no intermediate compile and link steps, like those typical for languages such as C or C++. Python executes programs immediately, which makes for an interactive programming experience and *rapid turnaround* after program changes—in many cases, you can witness the effect of a program change nearly as fast as you can type it.

Of course, development cycle turnaround is only one aspect of Python's ease of use. It also provides a deliberately simple syntax and powerful built-in tools. In fact, some have gone so far as to call Python *executable pseudocode*. Because it eliminates much of the complexity of its contemporaries, Python programs are simpler, smaller, and more flexible than equivalent programs in other popular languages.

Which is not to say that Python makes programming a *no-brainer*. Python also has convolutions and dark corners that we'll tackle head-on in this book, and some of its roles are more rapid than others. Python's flavor of OOP inheritance, for example, is much more complicated than it once was, and building standalone apps or precompiled programs of the sort you'll meet in the next chapter can still be slow. Measured by its peers, though, Python is dramatically more coder friendly.

It's Relatively Easy to Learn

Finally, this brings us to the point of this book: especially when compared to other widely used programming languages, the core Python language is remarkably easy to learn. In fact, if you're already an experienced programmer, you can expect to be coding small-scale Python programs in a matter of days—though you shouldn't expect to become an expert quite that fast, despite what you may have heard from marketing departments!

Naturally, mastering any topic as substantial as today's Python is not trivial, and we'll devote the rest of this book to this task. But the investment required to master Python is worthwhile: in the end, you'll gain programming skills that apply to nearly every computer application domain. Moreover, most find Python's learning curve to be much gentler than that of other programming tools, even if that curve is not quite as flat as some content publishers claim.

That's good news for both professional developers seeking to add the language to their toolbox, and end users of systems that expose a Python layer for scripting roles. Today, many systems rely on the fact that people can learn enough Python to use the system with little or no support. Moreover, Python has spawned a legion of users who program for need or fun instead of career and may never require

full-scale software development skills. Although Python has advanced tools you'll meet in this book, its core fundamentals are accessible to beginners and gurus alike.

To see all this for yourself, let's wrap up this overview and get started coding.

Chapter Summary

And that concludes the “hype” portion of this book. In this chapter, you've explored some of the reasons that people pick Python for their programming tasks. You've also seen how it is applied and looked at a representative sample of notable users today. This book's goal is to teach Python, though, not to sell it. The best way to judge a language is to see it in action, so the rest of this book focuses entirely on the language details glossed over here.

The next two chapters begin your technical introduction to the language. In them, you'll study ways to run Python programs, peek at Python's execution model, and learn the basics of module files for saving code. The aim will be to give you just enough information to run the examples and exercises in the rest of the book. You won't really start programming per se until [Chapter 4](#), but make sure you have a handle on the startup details before moving on.

Test Your Knowledge: Quiz

In this book, we will be closing each chapter with a quick open-book quiz about the chapter's coverage to help you review key concepts. The answers for these quizzes appear immediately after the questions, and you are encouraged to read the answers once you've taken a crack at the questions yourself, as they sometimes give useful summary context.

In addition to these end-of-chapter quizzes, you'll find lab *exercises* at the end of each *part* of the book, designed to help you start coding Python on your own, with suggested answers available in an appendix. For now, here's your first quiz. Good luck, and be sure to refer back to this chapter's material as needed.

1. What are the six main reasons that people choose to use Python?
2. Name four notable companies or organizations using Python today.
3. Why might you *not* want to use Python in an application?
4. What can you do with Python?

Test Your Knowledge: Answers

How did you do? Here are the suggested answers, though there may be multiple solutions to some quiz questions. Again, even if you're sure of your answers, you're encouraged to look at the suggestions for additional context. See the chapter's coverage for more details if any of these responses don't make sense to you.

1. Software quality, developer productivity, program portability, support libraries, component integration, and simple enjoyment. Of these, the quality and productivity themes seem to be the main reasons that people choose to use Python, but enjoyment counts, too, in a field that can be as challenging as software.

2. Google, Industrial Light & Magic, JPL, ESRI, Instagram, and many more. Almost every organization doing software development uses Python in some fashion, whether for long-term strategic product development or for short-term tactical tasks such as testing and system administration.
3. Python's main downside is performance: in its currently most-common version, at least, it won't run as quickly as fully compiled languages like C and C++. On the other hand, it's quick enough for most applications, and typical Python code runs at close to C speed anyhow because it invokes linked-in C code in the interpreter. If speed is critical, compiled extensions and other tools like Cython are available to optimize the number-crunching parts of a Python application.
4. You can use Python for nearly anything you can do with a computer, from website development and gaming to AI and spacecraft control. Numeric programming and web development may lead the pack today, though probably because those are some of the main things for which computers are used.

How Python Runs Programs

This chapter and the next cover program execution—how you launch code and how Python runs it. In this chapter, we'll begin by studying how the Python interpreter executes programs in general, from an abstract vantage point. With that perspective in hand, [Chapter 3](#) will dive into the nuts and bolts of getting your own programs up and running.

Startup details are inherently platform specific, and some of the material in these two chapters may not apply to the ways you'll be using Python, so you should feel free to skip parts not relevant to your goals. Likewise, readers who have used similar tools in the past and prefer to get to the meat of the language quickly may want to file some of these chapters away for future reference. For the rest of us, let's take a brief look at the way that Python will run our code, before we get into the details of writing or running it.

Introducing the Python Interpreter

So far, we've mostly been talking about Python as a programming language. In its most widely used form, though, it's also a software system called an *interpreter*. An interpreter is a kind of program that executes other programs. When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains. In effect, the interpreter is a layer of logic between your code and the computer hardware on your machine.

When Python is installed, it generates a number of components—minimally, an interpreter and a support library. Depending on how you use it, the Python interpreter may take the form of an executable program, or a set of libraries linked into another program. Depending on which flavor of Python you run, the interpreter itself may be implemented as a C program, a set of Java classes, or something else. Whatever form it takes, the Python code you write must always be run by this interpreter. And to enable that, you usually must install a Python interpreter on your computer.

You don't need to install Python at this point unless you want to work along with the sole trivial example coming up, and this book won't assume that you've got a Python ready to go until the next chapter. When you're ready, Python installation details vary by platform and are discussed briefly in [Chapter 3](#), and covered in full in [Appendix A](#).

Program Execution

What it means to run a Python script depends on whether you look at this task as a programmer, or as a Python interpreter. Both views offer important perspectives on Python programming.

The Programmer's View

In its simplest but most common form, a Python program is just a text file containing Python statements. For example, the file listed in [Example 2-1](#), named *script0.py*, may be one of the most trivial Python scripts one could dream up, but it passes for a fully functional Python program.

Example 2-1. script0.py

```
print('hello world')
print(2 ** 100)
```

This file contains two Python `print` statements, which simply print a string (the text in quotes) and a numeric expression result (2 to the power 100) to the output stream (the GUI or window where the file is run, normally). Don't worry about the syntax of this code yet—for now, we're interested only in how it runs. This book will explain the `print` statement, and why you can raise 2 to the power 100 so easily in Python, in its later chapters.

You can create such a file of statements with any text editor you like; see [Appendix A](#) for suggestions. By convention, Python program files are given names that end in *.py*; technically, this naming scheme is required only for files that are “imported”—a term clarified in the next chapter—but most Python files have *.py* names for consistency.

After you've typed these statements into a text file, you must tell Python to *execute* the file—which simply means to run all the statements in the file from top to bottom, one after another. As you'll see in the next chapter, you can launch Python program files by typing command lines, by clicking their icons, from within coding GUIs, and with other techniques. If all goes well, when you execute the file, you'll see the results of the two `print` statements show up somewhere on your computer—usually and by default, in the same window you were in when you ran the program:

```
hello world
1267650600228229401496703205376
```

For example, here's what happened when this script was run in a Command Prompt window with a command line on a Windows laptop, to make sure it didn't have any silly typos:

```
C:\Users\me\code> py script0.py
hello world
1267650600228229401496703205376
```

See [Chapter 3](#) for the full story on this process, especially if you're new to programming; we'll get into all the details of writing and launching programs there. For our purposes here, we've just run a Python script that prints a string and a number. We probably won't attract venture capital or go viral on GitHub with this code, but it's enough to capture the basics of program execution.

Python's View

The brief description in the prior section is fairly standard for scripting languages, and it's usually all that most new Python programmers need to know. You type code into text files, and you run those files through the interpreter. Under the hood, though, a bit more happens when you tell Python to “go.”

Although knowledge of Python internals is not strictly required for Python programming, having a basic understanding of Python’s runtime structure up front can help you grasp how your code fits into the bigger picture of program execution.

When you instruct Python to run your script, there are a few steps that Python carries out before your code actually starts crunching away. Specifically, it’s first compiled to something called “bytecode” and then routed to something called a “virtual machine.” This holds true only for the most common version of Python, and you’ll meet variations on this model in a moment. Since the most common is, well, most common, let’s see how this works first.

Bytecode compilation

Internally, and almost completely hidden from you, when you execute a program Python first compiles your *source code* (the statements in your text file) into a format known as *bytecode*. Compilation is simply a translation step, and bytecode is a lower-level, platform-independent representation of your source code. Roughly, Python translates each of your source statements into a group of bytecode instructions by decomposing them into individual steps. This bytecode translation is performed to speed execution—bytecode can be run much more quickly than the original source code statements in your text file.

You’ll notice that the prior paragraph said that this is *almost* completely hidden from you. If the Python program has write access on your machine, it will save the bytecode of your programs in files that end with a *.pyc* extension (*.pyc* means *.py* source, compiled). Technically, this save doesn’t happen when running a single file as we did in the preceding section but does for all but the topmost file in meatier multifile programs (more on this in a moment).

Python saves its bytecode files in a subdirectory named `__pycache__` located in the directory where your source files reside, and in files whose names identify the Python version that created them (e.g., *script0.cpython-312.pyc* for 3.12). The `__pycache__` subdirectory avoids clutter, and the naming convention for bytecode files prevents different Python versions installed on the same computer from overwriting each other’s saved bytecode. We’ll study this bytecode file model in more detail in [Chapter 22](#), though it’s automatic and irrelevant to most Python programs and is free to vary among the alternative Python implementations described ahead.

So why all the bother? In short, Python saves bytecode like this as a *startup-speed* optimization. The next time you run your program, Python will load the *.pyc* files and skip the compilation steps, as long as the bytecode is present, you haven’t changed your source code since the bytecode was last saved, and you aren’t running with a different Python than the one that created the bytecode. It works like this:

Source changes

Python saves the last-modified timestamp and size (or hash value, optionally) of a source code file in its bytecode file, and compares this info to the source when the bytecode is loaded to know when it must recompile—if you edit and resave your source code, its bytecode is re-created the next time your program is run.

Python versions

Python also adds a version-information suffix to bytecode filenames to know when it must recompile—if you run your program on a different Python implementation or version, its bytecode is generated and saved for that Python too.

The result is that both source code changes and differing Python versions will trigger a new bytecode file automatically. If Python *cannot* write the bytecode files to your machine, your program still works—the bytecode is generated in memory and simply discarded on program exit.

Because `.pyc` files speed startup time, though, you'll want to make sure they are written for larger programs. Bytecode files are also one way to ship Python programs—Python is happy to run a program if all it can find are compatible `.pyc` files, even if the original `.py` source files are absent. This isn't as simple as deleting the `.py` files, though, and may require file moves and renames, or special techniques discussed later in [Chapter 22](#). See Python's `compileall` module to force compiles when needed for packaging, and frozen binaries (see [“Standalone Executables” on page 24](#)) for another shipping option.

Strictly speaking, bytecode is an *import* optimization. Bytecode is saved in `.pyc` files only for files that are *imported*, not for the top-level files of a program that are only run as scripts. Moreover, a given file is imported and possibly compiled only *once* per program run; later imports use what's already been loaded. We'll explore import basics in [Chapter 3](#) and take a deeper look at imports in [Part V](#). For now, keep in mind that bytecode is never saved for code typed at the *interactive prompt*—a programming mode you'll learn about in [Chapter 3](#) and use early in this book.

The Python Virtual Machine (PVM)

Once your program has been compiled to bytecode (or the bytecode has been loaded from existing `.pyc` files), it is shipped off for execution to something generally known as the Python Virtual Machine (PVM, for acronym-inclined readers). The PVM sounds more impressive than it is; really, it's not a separate program, and it need not be installed by itself. In fact, the PVM is just a big code loop that iterates through your bytecode instructions, one by one, to carry out their operations. That is, the PVM is the runtime engine in Python. It's always present as part of the Python system, is the component that truly runs your scripts, and is really just the last step of the “Python interpreter.”

[Figure 2-1](#) illustrates this runtime structure. Bear in mind that all of this complexity is deliberately hidden from Python programmers. Bytecode compilation is automatic, and the PVM is just part of the Python system that you have installed on your machine. Again, programmers simply code and run files of statements, and Python handles the logistics of running them behind the scenes.

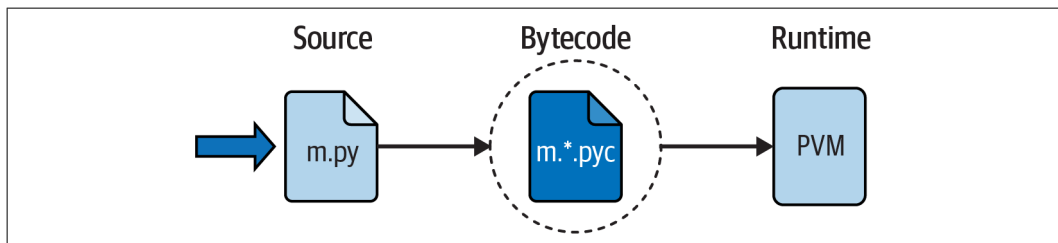


Figure 2-1. Python's traditional execution model: the PVM runs compiled bytecode

Performance implications

Readers with a background in fully compiled languages such as C and C++ might notice some glaring differences in the Python model. For one thing, there is usually no build or “make” step in Python work: code runs immediately after it is written. For another, Python bytecode is not binary *machine code* (e.g., instructions for an Intel or ARM chip, known as a *CPU*): it's a Python-specific format. There are exceptions to these rules (e.g., app builds for smartphones can take some time, and full compilers do exist, as you'll see ahead), but we're focusing on the common here first.

These differences explain why some Python code may not run as fast as C or C++ code, as described in [Chapter 1](#)—the PVM loop, not the CPU chip, still must interpret the bytecode, and bytecode instructions require more work than CPU instructions. On the other hand, unlike in classic interpreters, there is still an internal compile step—Python does not need to reanalyze and reparse each source statement's

text repeatedly. The net effect is that pure Python code runs at speeds somewhere between those of a traditional compiled language and a traditional interpreted language.

Development implications

Another ramification of Python's execution model is that there is really no distinction between the development and execution environments: the systems that compile and execute your source code are really one and the same. This similarity may have a bit more significance to readers with a background in traditional compiled languages, but in Python, the compiler is always present at runtime and is part of the system that runs programs.

This makes for a much more *rapid* development cycle. There is no need to precompile and link before execution can begin; simply type and run the code. This also adds a much more *dynamic* flavor to the language—it is possible, and often very convenient, for Python programs to construct and execute other Python programs at runtime. The `eval` and `exec` built-ins, for instance, accept and run strings containing Python program code. This structure is also why Python lends itself to product customization—because Python code can be changed on the fly, users can modify the Python parts of a system on-site without needing to compile or even possess the rest of the system's code.

At a more fundamental level, keep in mind that all we really have in Python is *runtime*—there is no initial compile-time phase at all, and everything happens as the program is running. This even includes operations such as the creation of functions and classes and the linkage of modules. Such events often occur before execution in more static languages, but happen during execution in Python. As you'll see, this makes for a much more dynamic programming experience than that to which some readers may be accustomed.

Why Does Python Use Bytecode?

Given that bytecode generally runs more slowly than machine code, this is a great question. The short answer is that Python uses bytecode for the sake of development speed and language flexibility.

In more detail, every program must ultimately run as machine code on the host device's CPU, but program code is just text written per a language's rules. Traditional languages like C bridge this gap by constraining code to accommodate the CPU's expectations and translating the code's text to machine code ahead of time. This makes programs fast, but translation takes time, and the resulting languages are cumbersome to use.

Python instead defines an easy-to-use language that's too far removed from machine code for a direct translation and uses the PVM intermediary to run your program's bytecode on the CPU. This is a classic speed-versus-usability trade-off, but also a false dichotomy: many of the alternative implementations you'll meet ahead do compile some Python code to machine code, and Python is quick enough for many roles even with its PVM model. For more on this trade-off, see [“OK, but What's the Downside?” on page 6](#).

Execution-Model Variations

Now that you've studied the internal execution flow described in the prior section, you should also know that it reflects just the standard implementation of Python today and is not a requirement of the Python language itself. Because of that, the execution model is prone to change with time. In fact, many

systems already modify the picture in [Figure 2-1](#) in one way or another. Before moving on, let's briefly explore the most prominent of these variations.

Python Implementation Alternatives

As of this writing, there are multiple implementations of the Python language. Although there is much cross-fertilization of ideas and work between these Pythons, each is a separately installed software system, with its own project and user base. All but one of these systems are optional reading for most Python beginners, but a quick look at the ways they modify the execution model might help demystify what it means to run code in general.

The short story is that *CPython* is the standard implementation—what we've called the “common” version so far. It's the usual Python on PCs and smartphones, and the system that most readers will be using (and if you're not sure, this probably includes you). This is also the version used in this book, though the Python language fundamentals presented here apply to all the alternatives too. All the other Python implementations have specific goals and may or may not implement the full Python language defined by CPython, but come close enough to qualify as Pythons.

For example, *PyPy* is a drop-in replacement for CPython that runs many programs quicker by compiling parts of them further as they run. *Jython* and *IronPython* instead reimplement CPython to provide access to Java and .NET components; although standard CPython programs can access such components too (e.g., via *pyjnius* and *Chaquopy* for Java), *Jython* and *IronPython* aim to be more seamless. Other options accelerate numeric code, or code in general.

In more detail, the following list is a quick rundown on the most prominent Python implementations available today—with the usual apologies to current options omitted for space, and future options omitted for lack of a crystal ball:

CPython: the standard

The original, standard, and reference implementation of Python is usually called CPython when you want to contrast it with the other options (and just plain “Python” otherwise). This name comes from the fact that it is coded in portable ANSI C language code. This is the Python that you fetch from python.org for PCs, get with most alternative distributions and Linux repos, and have inside Python apps on Android and iOS smartphones. This is also the flavor whose implementation is captured in [Figure 2-1](#), though this it prone to change (and in fact may: see “[Future Possibilities](#)” on page 25).

Jython: Python for Java

The Jython system is an alternative implementation of the Python language. It's targeted at integration with the Java programming language, much as CPython integrates with C and C++ components. Jython consists of Java classes that compile Python source code to Java bytecode, which is then routed to the Java Virtual Machine (JVM). Programmers still code Python statements in *.py* text files as usual; the Jython system essentially just replaces the rightmost two bubbles in [Figure 2-1](#) with Java-based equivalents for seamless Java linkage. At this writing, Jython implements the older Python 2.X not used in this book but is working toward 3.X.

IronPython: Python for .NET

IronPython is another alternative implementation of Python. Coded in C#, it is designed to allow Python programs to integrate with applications written to work with Microsoft's .NET Framework for Windows, as well as the Mono open source equivalent. Like Jython, IronPython replaces the last two bubbles in [Figure 2-1](#) with equivalents for execution in the .NET environment. With it,

Python code can gain accessibility both to and from other .NET languages and leverage .NET libraries.

Stackless: Python for concurrency

The Stackless Python system is an enhanced version of the standard CPython language oriented toward concurrency. Because it does not save state on the C language call stack, Stackless can make Python easier to port to small-stack architectures. Stackless also provides efficient multiprocessing options that some find more straightforward than CPython's later `async` coroutines, and fosters novel programming structures. As an example, the game *EVE Online* uses Stackless Python to achieve high performance for massively parallel tasks.

PyPy: a JIT compiler for speed

PyPy is a reimplement of Python for speed. It was one of the first systems to employ a just-in-time (JIT) compiler for normal Python code. A JIT compiler is just an extension to the PVM—the rightmost bubble in [Figure 2-1](#)—that translates portions of your bytecode all the way to machine code for faster execution. PyPy does this as your program is running, not in a prerun compile step, and is able to create type-specific machine code for the dynamic Python language by keeping track of the data types of the objects your program processes. By replacing portions of your bytecode this way, your program runs faster and faster as it is executing. In addition, some Python programs may also take up less memory under PyPy.

Numba: a JIT compiler for numeric speed

The Numba extension for Python adds a JIT compiler that optimizes numerically oriented code by compiling it all the way to machine code while your program runs. To direct Numba's compiler, functions are augmented with Python “@” decorators supplied with the Numba install. While not all Python code can be sped up by Numba, it works well for code that uses *NumPy* arrays and functions, as well as math-oriented loops. Numba also supports code parallelization paradigms commonly used in scientific programming.

Shed Skin: an AOT compiler for conforming code

Shed Skin is an ahead-of-time (AOT) compiler that translates unadorned Python code to C++ code, which is then compiled to machine code before it is run. With an AOT, the two rightmost bubbles in [Figure 2-1](#) are replaced with precompiled machine code. Shed Skin can yield both standalone programs and extension modules for use in other programs. In exchange, it implements a restricted subset of Python that requires Python variables to meet an implicit statically typed constraint and does not support some Python features or libraries today. Nevertheless, Shed Skin may outperform both CPython and JIT-based options for some conforming code.

PyThran: an AOT compiler for numeric speed

PyThran implements another AOT compiler for a subset of the Python language, with a focus on scientific programming. Like Shed Skin, it translates Python code to C++, using static type declarations provided in either formatted comments or separate command files. The result of compiling the generated C++ is a native module that can be imported and used by other Python code.

Cython: a Python/C hybrid for speed

The Cython system defines a Python/C hybrid language that combines Python code with the ability to call C functions and use C type declarations for variables, parameters, and class attributes. Cython code can be AOT-compiled to C code that uses the Python/C API, which may then be compiled completely to machine code. Though not compatible with standard Python, Cython can be useful both for wrapping external C libraries and for implementing performance-critical parts of a system as efficient C extensions for use in CPython programs.

MicroPython: a Python subset for constraints

The MicroPython system is an alternative Python with a focus on efficiency. It implements a limited dialect of the CPython language and a small subset of its standard library, in order to optimize Python to run in constrained environments. Though originally targeted at microcontrollers, MicroPython is also compiled for the *WebAssembly* system you'll meet in [Chapter 3](#), to support Python programs in web browsers without the full heft of CPython.

And (naturally) more

For other Python alternatives, see the list at [Python's wiki](#), as well as the results of a web search. Among the others, *Cinder* is a performance-focused implementation of CPython with a JIT compiler and more, created by Meta and used for Instagram; *Pyston* is a fork of CPython with a JIT compiler and other optimizations, started by Dropbox; and *Nuitka* is a free and paid optimizing AOT compiler that translates standard Python code to C, and is used by the emerging *py2wasm* to translate Python code to *WebAssembly* for use in browsers.

All that being said, unless you have a specific need met only by one of the alternatives, you'll probably want to use the standard CPython system. Because it is the reference implementation of the language, it's always the most complete, and also tends to be more up-to-date and robust than others. Still, it's unlikely that CPython will ever subsume all the optimization projects afoot in the Python world, so be sure to vet the alternatives when your goals gel.

Standalone Executables

Sometimes when people ask for a “real” Python compiler, what they're actually seeking is simply a way to generate standalone executables from their Python programs. This is more a packaging and shipping topic than an execution-flow concept—and of more interest to software developers than Python beginners—but it's a related idea.

In short, with the help of third-party tools that you can fetch off the web, it is possible to turn your Python programs into true self-contained executables, sometimes also called *frozen binaries* in the Python world. Whatever they're called, these programs can be run without requiring a Python installation or shipping your source code files.

Standalone executables bundle together the bytecode of your program files, along with the PVM (interpreter) and any Python support files and libraries your program needs, into a single package. There are some variations on this theme, but the end result can be a single *executable* (e.g., a *.exe* file on Windows) or *app* (e.g., a *.app* on macOS, and *.apk* or *.aab* on Android) that can easily be shipped to customers. In [Figure 2-1](#), it is as though the two rightmost bubbles—bytecode and PVM—are merged into a single component: a standalone executable bundle.

Today, a variety of systems are capable of generating standalone executables and vary in platforms and features. For example, *py2exe* builds standalones for Windows; *PyInstaller* and *cx_freeze* make them for Windows, macOS, and Linux; *py2app* creates them for macOS; and *Buildozer* and *Briefcase* generate apps for Android and iOS.

Frozen binaries are not the same as the output of a true compiler—they run bytecode through a virtual machine. Hence, apart from a possible startup improvement, frozen binaries run at the same speed as the original source files. Frozen binaries are also not generally small (they contain a PVM), but by current standards they are not unusually large either. Because Python is embedded in the frozen bundle, though, it does not have to be installed on the receiving end to run your program. Moreover, because your bytecode is embedded in the bundle, it isn't as easily viewed.

For more details, see the alternative coverage of standalones in this book's [Appendix A](#), which includes tips on building them on multiple platforms from the same code base.

Future Possibilities

Finally, note that the runtime execution model sketched here is really an artifact of the current implementation of Python, not of the language itself. For instance, it's possible that an AOT compiler for translating unrestricted and unadorned Python source code to machine code may appear during the shelf life of this book (although the fact that one has not in over three decades makes this seem unlikely!), and JIT compilers seem to be cropping up everywhere.

Either way, the bytecode model will likely be standard for some time to come. The portability and flexibility of bytecode are important features of many Python systems. Moreover, adding type-constraint declarations to support static compilation would break much of the flexibility, conciseness, simplicity, and spirit of Python coding we're about to explore. Python's type hinting, also covered later, comes close, but is thankfully still unused by Python itself today.



JIT futurism: While *CPython* currently follows the bytecode/PVM model in [Figure 2-1](#), it may augment it in the future. Version 3.13, still under development as this is being written, will add an experimental JIT compiler. As in *PyPy* and others, this will translate some bytecode all the way to machine code as your program is running. In 3.13 it will have a negligible speed boost and will be disabled by default. Though prescience is perilous in publishing, the JIT compiler may be enabled by default for *CPython* in the future if it ever yields a significant net gain for Python programs.

Chapter Summary

This chapter introduced the execution model of Python—how Python runs your programs—and explored some common variations on that model designed for both different roles and better performance. Although you don't really need to come to grips with Python internals to write Python scripts, a passing acquaintance with this chapter's topics will help you truly understand how your programs run once you start coding them, as you will in the next chapter. First though, here's the usual chapter quiz to review what you've learned so far.

Test Your Knowledge: Quiz

1. What is the Python interpreter?
2. What is source code?
3. What is bytecode?
4. What is the PVM?
5. What is machine code?
6. Name two or more variations on Python's standard execution model.
7. How are *CPython*, *Jython*, and *IronPython* different?
8. What are *PyPy*, *Shed Skin*, and *Cython*?

Test Your Knowledge: Answers

1. The Python interpreter is a program that runs the Python programs you write. It essentially intermediates between your Python instructions and those available in a CPU's machine code.
2. Source code is the statements you write for your program. It consists of text in text files whose names normally end with a *.py* extension.
3. Bytecode is the lower-level form of your program after Python compiles it. Python automatically stores bytecode in files with a *.pyc* extension when possible, and automatically re-creates it when needed.
4. The PVM is the Python Virtual Machine—the runtime engine of Python that interprets your compiled bytecode.
5. Machine code is the low-level instructions of the underlying CPU on a computing device like a PC or phone. Because this is what every program ultimately runs, Python code must be translated to this by a software layer like the PVM interpreter, or JIT or AOT compilers.
6. Numba, Shed Skin, and standalone executables are all variations on the execution model. In addition, the alternative implementations of Python named in the next two answers modify the model in some fashion as well—by replacing bytecode and VMs, or by adding JIT and AOT compilers.
7. CPython is the standard and reference implementation of the language. Jython and IronPython process Python programs for use in Java and .NET environments, respectively; they are alternative compilers for Python.
8. PyPy and Shed Skin are reimplementations of Python targeted at speed. PyPy speeds normal Python programs by using runtime type information and a JIT compiler to replace some Python bytecode with machine code as the program runs. Shed Skin speeds programs with an AOT compiler that translates a restricted subset of Python to C++, from which it can be fully compiled to machine code to be run as a program or used in others. Cython is a Python/C combination that can be compiled into machine-code extensions accessible to CPython code.

How You Run Programs

OK, it's time to start running some code. Now that you have a handle on Python's purpose and execution model, you're finally ready to start some real Python programming.

There are multiple ways to tell Python to execute the code you type, and this chapter covers all the major program launching techniques in common use today. Along the way, you'll learn both how to run code *interactively*, and how to save it in *files* to be run using a variety of techniques—with command lines, icon clicks, the IDLE GUI, mobile apps, web-based interfaces, module imports, and more.

As for the previous chapter, if you have prior programming experience and are anxious to start digging into Python itself, you may want to skim this chapter and move on to [Chapter 4](#). But don't skip this chapter's early coverage of preliminaries and conventions, its overview of debugging techniques, or its first look at module *imports*—a topic essential to understanding Python's program architecture, which we won't revisit until [Part V](#). It's also worthwhile to browse the sections on IDEs and apps, to sample tools that may be more useful once you start coding larger programs.

Installing Python

This book will generally assume that you have access to a recent version of Python on your computer, tablet, or phone. Python installation is not required for this book, and isn't necessary in some contexts, but you'll need a Python to work along with examples and do end-of-part exercises, and both are great ways to make concepts more concrete.

If you don't have a Python and wish to set one up, see [Appendix A](#) now for per-platform install help. In short:

- *Windows* and *macOS* users fetch and run a self-installing executable file that puts Python on their devices. Simply double-click and say Yes or Next at all prompts.
- *Linux* (including Windows *WSL*) users may have a usable Python preinstalled on their computers, but can install one if needed or desired from their distribution's repositories.
- *Android* and *iOS* users install an app that allows them to run Python locally on their phones and tablets.
- *Unix* (and some Linux) users often compile Python from its full source code distribution package.

For example, smartphone users fetch a Python app at an app store, and Windows and macOS PC users get an installer at the [downloads page](#) of Python's main website. Python may also be had through other distribution channels, and some Python coding modes covered ahead, such as *Jupyter* notebooks, have unique install and usage steps. You may not be able to select a Python version in some contexts, but this book uses Python 3.12, so closer to that is better.

Before you install, you should generally check to see if Python is already present. Look for it on Windows in the Start menu, and on most platforms by running a Python command line as described both in [Appendix A](#) and this chapter's next section. To see how, let's take installs as a given, and move on to learning the many ways to run Python code.

Interactive Code

This section gets us started with the basics of interactive Python coding. Because it's our first look at running code, we also cover some startup logistics here, such as setting up a working directory, so be sure to read this section first if you're relatively new to programming. This section also explains some conventions used throughout the book, so most readers should probably take at least a quick look here.

Starting an Interactive REPL

By most measures, the simplest way to run Python code is to type it at Python's interactive command line, sometimes called the *interactive prompt*, and often more concisely labeled the *REPL* (which stands for read-eval-print loop).

There are a variety of ways to start this command line—in a coding GUI or app, in a web notebook, from a system console, and so on. Assuming Python is installed locally as an executable program on your device, the most platform-neutral way to start an interactive interpreter session is to simply type a *Python command* at your device's console prompt, without any arguments. Since this is most common, nearly universal, and arguably simplest, let's start here.

Despite this scheme's generality, both the Python command you'll type and where you'll type it vary per platform. On *Windows*, *macOS*, and *Android*, for example, `py`, `python3`, and `python`, respectively, do the job as follows:

```
$ py
Python 3.12.3 (...etc..., Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z

$ python3
Python 3.12.2 (...etc..., Feb  6 2024, 17:02:06) [Clang 13.0.0...] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D

$ python
Python 3.11.4 (...etc..., Jul  2 2023, 11:17:00) [Clang 14.0.7...] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D
```

A `python3` works on *Android* and *Linux*, too, if you want to narrow this to one exception—for Windows' `py`. This book uses `python3` in platform-neutral examples just because it works almost everywhere. If you're working on Windows, instead use `py` (or `py -3` to ensure Python 3.X); `python3` works on Windows, too, but is reserved for a lesser Store install today per [Appendix A](#). Also per that appendix,

all these commands must be on your system search path (generally known as `PATH`) and require a full pathname if they're not, but most installs set this up automatically.

Typing a Python command at your system prompt like this begins an interactive Python session (i.e., REPL). The `$` character at the start of listings here stands for a generic system prompt on all platforms throughout this book—it will likely vary on your device and is not input that you type yourself. On *Windows*, a Ctrl+Z key combo (followed by Enter) ends this session as shown; on *Unix* (which includes macOS, Linux, and Android), it's Ctrl+D instead.

You'll notice that the interactive prompt opens with a message that identifies the Python being used (e.g., “3.12.3” is Python 3.12) and the platform it's running on (e.g., “win32” for Windows, “darwin” for macOS, and “linux” for both Linux and Android in Python 3.12), followed by a line with tips for more info. We're going to omit these opening lines in this book, except where they're helpful.

The notion of a system prompt (a.k.a. *shell* or *console*) where you type the Python command is generic, but exactly how you access it varies by platform. It might be *Command Prompt* or *PowerShell* on Windows; *Terminal* on macOS and Linux; the *Termux* app on Android; or a dedicated terminal screen in some apps like *Pydroid 3*.

On most platforms, you can also start an interactive session in ways that don't require typing a command at all, but they vary even more widely. On *Windows*, for example, a Start-menu option opens a similar REPL, as do the IDLE GUI on all PCs, dedicated screens in apps on Android and iOS, and some web-browser interfaces. We'll cover some of these options ahead, but see [Appendix A](#) for more platform tips, and the web for more help with other options.

Anywhere you see the `>>>` prompt, though, you're in an interactive Python session and REPL—you can type any Python statement or expression here and run it immediately by simply pressing the Enter key (or similar button). We will do so in a moment, but first we need to get a few admin details sorted out to make sure all readers are set to go.

Where to Run: Code Folders

Now that you're starting to learn *how* to run code, you'll also need to know *where* to run code. You can save and run code anywhere you can make files, but this book has two recommendations for using its examples, especially for newcomers:

Work in a dedicated code folder

To avoid stomping on other content, run this book's examples from a dedicated code *folder* (a.k.a. directory) on your device. For instance, this book runs all its code in a folder nested in the user account (or “home”) folder on each device. Your code folder can be located wherever you like and called whatever you wish, but running out of one folder will help you keep track of your work and simplify some tasks.

Work in per-chapter subfolders of your code folder

To avoid clutter and filename collisions, also organize your code into per-chapter *subfolders*, nested in your dedicated code folder. Per-chapter subfolders will ensure that *imports* in examples will work without advanced settings (that you'll learn later). There's more on imports and directories in an upcoming note. For now, keep in mind that console commands in this book will implicitly be run in a per-chapter subfolder of a dedicated code folder on the host. The folder name will normally be omitted for platform neutrality and is irrelevant to the code.

If you'll be using this book's *examples package*, it's already done the setup work for you. For example, unzipping the examples creates a code folder named *LP6E*, whose *Chapter03* subfolder has this chapter's code. If you'll be using this package to avoid typing code or copying from emedia, simply run its examples in the subfolder of the chapter you're studying. This is where REPLs will be started and where script files will be run. As you'll learn later, this is also where *data files* that our scripts create will show up, unless scripts use filenames with explicit folder paths.

If you'll be creating code on your own, you should probably *create* a similar code directory structure of your own before we move on. On PCs, use your system's file explorer, or run a command line: `mkdir folder` works on Windows and Unix. On Android, you can also use a file explorer (or `mkdir` in Termux), but be sure to pick a folder accessible to your Python app. Some coding interfaces may offer other ways to create folders; see your tool for info.

Once your per-chapter code folders are set to go, always *start* there to write, save, and run the examples in this book. How you'll do this depends on your usage mode. In consoles, a portable `cd folder` command changes directories. In a GUI like IDLE, opening and running a file may go to its folder. And in other interfaces, you might launch a REPL for a file in the UI or use other schemes too varied to cover here. As a fallback, Python's `os.getcwd()` you'll meet shortly shows the current directory, and its `os.chdir('path')` changes it—as long as you import `os` first.

What Not to Type: Prompts and Comments

Speaking of commands, remember not to type the `$` character used at the start of this book's command lines to denote a system console prompt; type just the text *after* these prompts. This may sound simple to experienced programmers, but it's a very common first error for beginners, and we're not excluding anyone here.

Similarly, do not type the `>>>` and `...` prompt characters shown at the start of lines in interpreter interaction listings and used by this book to denote code run in a REPL; type just the text *after* these prompts. These are prompts that Python's standard REPL displays automatically as visual guides for interactive code entry and may or may not appear in your interface. For instance, the `...` prompt is used for *continuation lines* in some REPLs, but is just a label in IDLE, and is omitted by some of this book's listings for easier copy/paste; either way, don't type it yourself.

To help you remember this, user inputs you must type are shown in **bold** in this book, and prompts are not. Also keep in mind that commands typed after these system and Python prompts are meant to be run immediately and are not generally intended to be saved in the source files we will be creating; you'll see why this distinction matters ahead.

In the same vein, you normally don't need to type text that starts with a `#` character in this book's code listings—as you'll learn later, these are *comments*, not executable code. Except when `#` is used to introduce a `#!` directive at the top of a script, you can safely omit it and the rest of the line that follows it (there's more about `#!` in [Appendix A](#)).

Other Python REPLs

Having said all that, you should also know that Python REPLs can also be had in systems that convolute the traditional model covered in this chapter. *IPython*, for example, provides an alternative, separately installed, and enhanced Python interactive session, which labels commands by number and doesn't use `>>>` prompts. *Jupyter* notebooks provide the IPython REPL, too, and run it in a web browser instead of system console (we'll explore Jupyter later in this chapter).

To muddy this story further, the *PyPy* system of [Chapter 2](#) uses `>>>>` for its REPL prompts to distinguish it from CPython's ternary `>>>` (though you may have to look hard to tell); the *IDLE* GUI covered ahead displays `>>>` off to the side (and not in a system console); smartphone *apps* may vary too (see [Appendix A](#)); and the interactive prompt can technically be changed to anything (it's `sys.ps1` to your code, and can be set in a startup file, per Python docs).

All of which means that your Python interactive session may differ from the mainstream model that this chapter often employs. Depending on your tools and goals, you might see a different prompt or none at all, and you might type interactive Python code into a web browser, GUI, or app instead of the system console.

In general, though, this book recommends the traditional and simpler options it demos and covers, when you're first starting out. IPython and Jupyter, for example, have learning curves of their own, and Jupyter is geared toward scientific work, which is just one of many Python roles. If and when you opt to use alternative coding interfaces like these, and others sure to arise in the future, extrapolate to their REPLs' minor differences as needed.



REPL futurism: The standard CPython is scheduled to improve its interactive REPL interface in version 3.13, not yet released as this book was written. Per plans, the REPL will gain color prompts, automatic indentation, multiline editing and paste, and colorized exceptions borrowed from other UIs and IDEs. Colors won't matter in this book's print version, of course, and the future is not yet written, but a more colorful future may have arrived by the time you read this note.

Tip: Setting environment variable `PYTHON_COLORS` to 0 disables the new REPL colors if you find that future distracting.

Running Code Interactively

With those preliminaries out of the way, let's move on to typing some actual code. However it's started, the Python interactive session begins by printing some informational text (again, mostly omitted hereafter), then prompts for input with `>>>` (or similar) when it's waiting for you to type a new Python statement or expression.

When working interactively, the results of your code are displayed below the `>>>` input lines after you press the Enter key (or similar). For instance, here are the results of two Python `print` statements:

```
$ python3
>>> print('Hello world!')
Hello world!
>>> print(2 ** 8)
256
```

There it is—we've just run some Python code (it's not much, but it proves the point). We're still skipping most code details for now, but in brief, `print` is a built-in tool that sends a line displaying whatever you pass to it, to wherever you're working; because it's a *function call* in Python, the parentheses in this code are required. We've used it here to display a Python string and an integer, as shown by the output lines that appear after each `>>>` input line.

When coding interactively like this, you can type as many Python commands as you like; each is run immediately after it's entered. Moreover, because the interactive session automatically prints the results

of expressions you type, you don't usually need to say “print” explicitly at this prompt; the format of the following automatic displays can differ slightly from `print`, but it's not yet important to know how:

```
>>> language = 'Python'
>>> language
'Python'
>>> 2 ** 8
256
>>> ^Z                               # Use Ctrl-D (on Unix) or Ctrl-Z (on Windows) to exit
$
```

Here, the first line saves a value by assigning it to a *variable* (`language`), which is created by the `=` assignment; and the last two lines typed are *expressions* (`language` and `2 ** 8`), whose results are displayed automatically. Again, to exit an interactive session like this and return to your system prompt, type Ctrl+D on Unix-like machines, and Ctrl+Z on Windows. In the IDLE GUI discussed later, either type Ctrl+D everywhere, or simply close the window.

Notice the *italicized note* about this on the right side of this listing—starting with `#` here. This book uses these throughout to add remarks about what is being illustrated, but you don't need to type this text yourself; if you do, they're ignored by Python as *comments*. In fact, much like system `$` and Python `>>>` prompts, you shouldn't type this when it's on a system command line; the `#...` part is ignored by Python but may be an error in system shells.

Now, we didn't do much in this session's code—just typed some Python `print` and assignment statements, along with a few expressions, all of which we'll study in detail later. The main thing to notice is that the Python REPL executes the code entered on each line immediately, when the Enter key (or similar) is pressed.

For example, when we typed the first `print` statement at the `>>>` prompt, the output (a Python string) was echoed back right away. There was no need to create a source code file, and no need to run the code through a compiler and linker first, as you'd normally do when using a language such as C or C++. Strictly speaking, interactive code is compiled to bytecode in memory and run by the PVM in CPython (see [Chapter 2](#)), but you don't need to care.

As you'll see in later chapters, you can also run *multiline statements* (e.g., for loops) at the interactive prompt; such a statement may prompt for continuation lines with `...` as noted earlier and runs immediately after you've entered all of its lines and pressed Enter *twice* to add a blank line. Blank lines aren't required (and are ignored) in code files, but are needed to let some REPLs know your statements are complete. Also, bear in mind that the current and standard REPL runs just *one* statement at a time—don't paste large code blocks at its prompt!

Why the Interactive Prompt?

The interactive prompt runs code and echoes results as you go, but it doesn't save your code in a file. Although this means you won't do the bulk of your real-world coding in interactive sessions, the interactive prompt turns out to be a great place to both learn the language and test program files on the fly. Here's a quick rundown of these roles.

Learning

Because it executes code immediately, the interactive prompt is ideal for experimenting with code and learning the language, and we'll be using it throughout this book to demonstrate smaller examples and amplify concepts. In fact, this is the first rule of thumb to remember: if you're ever in doubt about how

a piece of Python code works, fire up the interactive command line and try the code out to see what happens.

For instance, suppose you're reading a Python program's code and you come across an expression like `'Hack!' * 8` whose meaning you don't understand. At this point, you can spend 10 minutes wading through manuals, books, and the web to try to figure out what the code does, or you can simply run it interactively:

```
$ python3
>>> 'Hack!' * 8                                     # Learning by trying
'Hack!Hack!Hack!Hack!Hack!Hack!Hack!Hack!'
```

This immediate feedback you receive at the interactive prompt is often the quickest way to deduce what a piece of code does. Here, it's clear that it does string repetition: in Python, `*` means multiply for numbers, but repeat for strings—it's like concatenating a string to itself repeatedly (there's more on strings in [Chapter 4](#)).

Chances are good that you won't break anything by experimenting this way—at least, not yet. To do real damage, like deleting files and running shell commands, you must really try by importing modules explicitly (we'll sample tools that can make you that dangerous later in this chapter). Straight Python code, though, is almost always safe to run.

For instance, watch what happens when you *make a mistake* at the interactive prompt:

```
>>> X                                               # Making mistakes is OK
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'X' is not defined
```

In Python, using a variable before it has been assigned a value is always an error—otherwise, if names were filled in with defaults, some errors might go undetected until it's too late. This means you must initialize counters to zero before you can add to them, must initialize lists to empty before extending them, and so on. You don't need to declare variables in Python, but they must be assigned before you can fetch their values.

Other error messages try to be more helpful in Python today with “Did you...?” tips—as when looking for the `sys.ps1` prompt hook mentioned earlier. You must import modules like `sys` before using them (though you probably won't need the reminder fairly soon, and we'll be truncating some error messages in this book for brevity):

```
>>> sys.ps1                                         # Requires "import sys"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sys' is not defined. Did you forget to import 'sys'?
```

You'll learn more about all that later. The important point here is that you won't *crash* Python or your computer when you make a mistake this way. Instead, you get a meaningful error message pointing out the mistake and the line of code that made it, and you can continue on in your session or script. In fact, once you get comfortable with Python, its error messages may often provide as much debugging support as you'll need (watch for more about debugging options in the sidebar [“Debugging Python Code” on page 49](#)).

Testing

Besides serving as a tool for experimenting while you're learning the language, the interactive interpreter is also an ideal place to test code you've written in files. You can import your module files interactively and run tests on the tools they define by typing calls at the interactive prompt on the fly.

For instance, the following tests a function in a precoded module that ships with Python in its standard library (detail: `os.getcwd` prints the name of the directory you're currently working in, here on a macOS host), but you can do the same once you start writing module files of your own:

```
>>> import os
>>> os.getcwd()                               # Testing on the fly
/Users/me/code'
```

More generally, the interactive prompt is a place to test program components, regardless of their source—you can import and test functions and classes in your Python files, type calls to linked-in C functions, exercise Java classes under Jython, and more. Partly because of its interactive nature, Python supports an experimental and exploratory programming style you'll find convenient. Although Python programmers also test with in-file code (and you'll learn ways to make this simple later in the book), for many, the interactive prompt is still their first line of testing defense.

Program Files

Although the interactive prompt is great for experimenting and testing, it has one big disadvantage: programs you type there go away as soon as the Python interpreter executes them. Because the code you type interactively is never stored in a file, you can't run it again without retyping it from scratch. Cut-and-paste and command recall can help some here, but not much, especially when you start writing larger programs. To cut and paste code from an interactive session, you would have to edit out Python prompts, program outputs, and so on—which is too tedious to try.

To save programs permanently, you need to write your code in *files*, which are usually known as *modules*. Modules are simply text files containing Python statements. Once they are coded, you can ask the Python interpreter to execute the statements in such a file any number of times, and in a variety of ways—by system command lines, by file icon clicks, by options in the IDLE user interface, and more. Regardless of how it is run, Python executes all the code in a module file from top to bottom each time you run the file.

Terminology in this domain can vary by role. For instance, module files are often referred to as *programs* in Python—that is, a program is considered to be a series of precoded statements stored in a file for repeated execution. Module files that are run directly are also sometimes called *scripts*—an informal term usually meaning a top-level program file. Some reserve the term *module* for a file imported from another file, and *script* for the main file of a program; we generally will here too (stay tuned for more on the meaning of top-level, imports, and main files later in this chapter).

Whatever you call them, the next few sections explore ways to run code typed into files. In this section, you'll learn how to run files in the most basic and portable way: by listing their names in a Python *command line* entered at your computer's system prompt. Though this might seem primitive to some—and can often be avoided altogether by using alternatives discussed later—for many programmers, a system console window for command lines, together with a text editor window, constitutes as much of an integrated development environment as they will ever need and provides more direct control over programs.

A First Script

Let's get started. Open your favorite text editor, type or copy/paste the statements in [Example 3-1](#) into a new text file named *script1.py*, and save it in your working code directory that you set up earlier (make it now if you skipped over that step). Any editor will work, including vi, Notepad, a smartphone app's

editor, and the IDLE GUI coming up soon. You can also find this file in the book examples package, but typing code is an important exercise early on.

Example 3-1. *script1.py*

```
# A first Python script
import sys                # Load a library module
print(sys.platform)
print(2 ** 100)           # Raise 2 to a power
x = 'Hack!'
print(x * 8)              # String repetition
```

This file is our first official Python script (not counting the two-liner in [Chapter 2](#)). You shouldn't worry too much about this file's contents just yet, but as a brief description, its code:

- Imports a Python module (libraries of additional tools) to fetch the name of the platform
- Runs three `print` function calls to display the script's results
- Uses a variable named `x`, created when it's assigned, to hold on to a string object
- Applies various object operations that we'll begin exploring in the next chapter

The `sys.platform` here is just a string that identifies the kind of computer you're working on; it lives in a Python module called `sys` (part of its standard library), which you must import to load (again, more on imports later). Also notice how this file uses explicit `print` calls; unlike the REPL, output in files is never automatic, so you must say `print` in files to see their output (and forgetting this is a regular first mistake, but at least you've been warned!).

For color, this file adds some Python *comments*—the text after the `#` characters. These were mentioned earlier, but should be more formal now that they're showing up in scripts. Comments can show up on lines by themselves, or to the right of code on a line. The text after a `#` is simply ignored as a human-readable note and is not considered part of the statement's syntax. If you're copying this code, you can ignore the comments; they are just informative. This book uses a different formatting style to make comments more visually distinctive, but they'll be normal text in your code.

Again, don't focus on the syntax of the code in this file for now; you'll learn about all of it later. The main point to notice is that you've typed this code into a file, rather than at the interactive prompt. In the process, you've coded a fully functional Python script.

Notice, though, that the module file is named *script1.py*. As for all *top-level* files (i.e., files run directly), it could also be named simply *script1*, but files of code you want to *import* in another file or REPL have to end with a *.py* suffix. Because you may want to import them in the future, it's a good idea to use *.py* suffixes for most Python files that you code. Also, some text editors and file explorers detect Python files by their *.py* suffix; if the suffix is not present, you may not get features like syntax colorization and automatic indentation in editors or tap-to-run in explorers.

Running Files with Command Lines

Once you've saved the preceding section's text file, you can ask Python to run it by listing its full filename as the first argument to a Python command—like the following typed at the system shell's `$` prompt on a Unix device (but don't type this at a Python REPL prompt, and read on to the next paragraph if this doesn't work right away for you):

```
$ python3 script1.py
darwin
1267650600228229401496703205376
Hack!Hack!Hack!Hack!Hack!Hack!Hack!Hack!
```

Just as for starting the REPL we studied earlier, the command name may vary (e.g., use `py` instead of `python3` on Windows, usually), and you can type such a command in whatever your system provides for command-line entry—*Command Prompt* on Windows, *Terminal* on macOS, or the *Termux* app on Android, among others. You might also run this code file with a dedicated run button in a GUI, app, or browser-based UI, but we'll postpone such options until later in this chapter (see also [Appendix A](#) for all things platform specific).

Be sure to run your Python command in the same working directory where you've saved your script file (`cd` there first if needed), and run it at the system prompt, not Python's `>>>` prompt. Also like starting the REPL, you may have to replace the command's first word with a full directory path if Python isn't on your `PATH` setting, but most installs automatically ensure that it is (see [Appendix A](#) for more on Python installs and `PATH`).

If all works as planned, this shell command (or similar) makes Python run the code in this file line by line, and you will see the output of the script's three `print` statements—the name of the underlying platform as known to Python, 2 raised to the power 100, and the result of the same string repetition expression we saw earlier (there's more on the meaning of the last two of these in [Part II](#)).

If all *didn't* work as planned, you'll get an error message—make sure you've entered the code in your file exactly as shown, and try again. We'll talk about debugging options in the sidebar “[Debugging Python Code](#)” on [page 49](#), but at this point in the book your best bet is probably rote imitation. And if all else fails, you might also try running under the IDLE GUI discussed ahead—a tool that sugarcoats some launching details, though sometimes at the expense of the more explicit control you have when using command lines.

If copying code grows too tedious or error-prone, you can also fetch this book's examples on the web, though again, typing code initially will help you learn to avoid syntax errors. See the [Preface](#) for info on obtaining the examples.

Command-Line Usage Variations

When you type a command to run a Python code file, the command you type is run by a system *shell* program (e.g., *Bash* on Unix). Because of this, all of the shell's syntax is available for more custom runs. For instance, you can route the printed output of a Python script to a file to save it for later use or inspection, by using special shell syntax:

```
$ python3 script1.py > saveit.txt
```

In this case, the three output lines shown in the prior run are stored in the file *saveit.txt* instead of being printed. This is generally known as *stream redirection*; it works for both output (`>`) and input (`<`) text and is available on Windows and Unix-like systems. This is useful for testing, as you can write programs that watch for changes in other programs' outputs. It also has little to do with Python, though (Python simply supports it), so we will skip further details on shell redirection syntax here. Redirection is for command lines only, though, because it's a function of the system shell.

On *Windows*, you can also type just the *name* of your script and omit the name of Python itself. Because Windows uses filename associations to find a program with which to run a file, the file's name is enough to run a *.py* file. The following command, for example, will automatically be run by Python on Windows (technically, by Python's *py* Windows launcher described in [Appendix A](#)):

```
$ script1.py
```

This works just as though you had clicked on the file's icon in File Explorer (a launch mode covered later). One fine point here: Command Prompt runs programs this way in its own window, but PowerShell may not; use `py` to view unredirected output in the latter if needed (or use the icon-click `input()` trick also coming up).

Finally, remember to give the full path to your script file if it lives in a different directory than the one in which you are working. For example, the `py` Python command in the following, run in PowerShell on Windows, assumes Python is in your system path but runs a file located elsewhere:

```
PS C:\Users\me\code> cd D:\temp\savecode           # Go to a different folder
PS D:\temp\savecode> py C:\Users\me\code\script1.py # Run a script elsewhere
win32
1267650600228229401496703205376
Hack!Hack!Hack!Hack!Hack!Hack!Hack!Hack!
```

If your `PATH` doesn't include Python's directory, and neither Python nor your script file is in the directory you're working in, use full paths for *both*—like the following on macOS, which for good measure throws in output stream redirection to a file located outside the *current working directory* (where you are when this command is run):

```
$ /usr/local/bin/python3 /Users/me/code/script1.py > /Users/me/data/saveit.py
```

This is a lot to type, but also pathological and atypical. To keep your commands simpler than this, make sure Python is on your `PATH`, and `cd` to script or data folders first. Again, most installs set up `PATH` automatically, so you need only focus on script and data folders when running commands.

Other Ways to Run Files

If you're not a fan of command lines, you can generally avoid them by launching Python scripts with file icon clicks, development GUIs, and other schemes that vary per platform and role. This book generally recommends command-line usage for learners, both because it's simple, and because it's a common, general, portable, and powerful way to run code. But it's not required. While the Python world is too rich in options for exhaustive coverage here, let's take a quick tour of the most prominent command-line alternatives to close out this chapter.

Clicking and Tapping File Icons

On most PC platforms, Python program files can be run by simply clicking or tapping their filename or icon in the local file explorer. For example, this works automatically on *Windows* in File Explorer, thanks to filename associations set up during Python's install. Clicks also run code files on *macOS* in Finder, if made to Open With the Python Launcher included in the macOS install; drags to the Python Launcher app when available work the same as clicks.

The file-clicks story is more involved on *Linux*, where files likely need executable permission and a `#!` first line to name Python. On smartphones, tapping a filename in a file explorer on *Android* (or Files on *iOS*) may open the file in an associated Python app, but this may work only for some explorer/app combos, and may not give access to all of a program's files in some contexts. For more insight, consult your platform and app docs, or experiment on your device.

Also see [Appendix A](#) for more info. As noted there, Windows and Linux clicks don't keep the window open for viewing output and error messages after programs end: if a script just prints and exits, it, well, just prints and exits—the console window appears, and text is printed there, but the console window

closes and disappears on program exit. Per [Appendix A](#), coding a closing call to Python's `input()` forces a pause before exit so you can see output, but this doesn't help for error messages. Use other run techniques when this matters.

The IDLE Graphical User Interface

So far, we've seen how to run Python code with interactive sessions, system command lines, and icon clicks. If you're looking for something a bit more visual, *IDLE* provides a *GUI* for Python programming, and it's a standard and free part of the Python system. IDLE is usually referred to as an *integrated development environment* (IDE), because it binds together multiple development tasks into a single view.

In short, IDLE lets you edit, run, browse, and debug Python programs, all from the same GUI. Because it's coded in Python with the tkinter GUI toolkit, it runs portably on all Python PC platforms—*Windows*, *macOS*, and *Linux*. For many, IDLE represents an easy-to-use alternative to typing command lines, a less problem-prone alternative to clicking on icons, and a great way for newcomers to get started editing and running code. You'll sacrifice some control in the bargain, but this typically becomes important only later in your Python career.

IDLE install and startup are covered in [Appendix A](#), so we won't repeat the full details here. In brief, it's standard with the *python.org* Python installers for Windows and macOS and can be had separately in Linux repositories. Once installed, it can be launched with the usual suspects: *Start* on Windows, *Launchpad* or *Finder* on macOS, a command line on Linux, and file right-clicks where supported.

[Appendix A](#) also has screenshots that capture IDLE in action on each platform, but [Figure 3-1](#) captures its dark theme on Windows for both variety and a quicker look; as it demos, after running this chapter's *script1.py* from [Example 3-1](#) in the edit window on top, its output appears in the interactive Shell window.

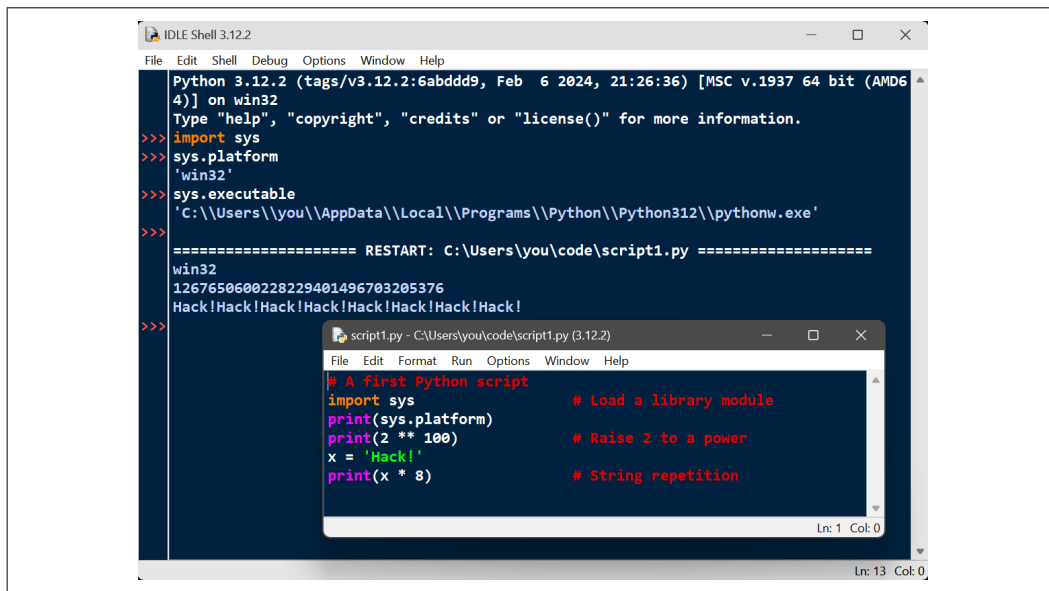


Figure 3-1. IDLE with its dark theme on Windows

Tip: because IDLE is just a Python script on the module search path in the standard library, you can also generally run it on any platform and from any directory by typing the following Python command in a system console window (use `py` instead of `python3` on Windows, as usual). Python's `-m` flag simply locates a module using the normal import search, but runs it as a top-level script ([Part V](#) covers both this search, and the `"."` package syntax required here):

```
$ python3 -m idlelib.idle           # Find and run idle.py in a package folder
```

Once IDLE is started, its usage is straightforward and documented in its in-program Help. In brief, the Shell window provides the usual interactive REPL with command recall, and each edit window allows you to view, modify, and run a file of code. The Shell's *File*→*New File* and *File*→*Open...* start and open code files. Edit windows' *Run*→*Run Module* runs code in the window where it's selected, and *Run*→*Run...* *Customized* supports passing command-line arguments to scripts (which is out of scope here and not a full system shell, but useful for scripts that expect these).

IDLE has many more features, but as a sampler: it also does tab completions, pop-up balloon help for functions, and object attribute lists as you type code, and includes an object browser and GUI debugger. To use IDLE's *debugger*, enable it in the Debug menu, set breakpoints by right-clicks in edit windows, and run. For simpler debugging needs, right-click on the text of any *error message* in the Shell window to jump to the line of code where the error occurred.

For more IDLE tips, see this book's [Appendix A](#), IDLE's own Help menu, and the notes for your platform in "Python Setup and Usage" in Python's standard manuals. Like most GUIs, the best way to learn IDLE may be to test-drive it for yourself. At the end of the day, its usability may be essential for some beginners, but it comes with an extra learning curve, is not as flexible as command lines, and adds sugarcoding that might be a negative when your programming needs outgrow its scope. As always, vet for yourself on a PC near you.

Other IDEs for Python

Because IDLE is free, portable, and a standard part of Python, it's a nice first development tool to become familiar with if you want to use an IDE at all. There are, however, a handful of alternative IDEs for Python developers, some of which are substantially more powerful and robust than IDLE.

Among these, *PyCharm*, *PyDev*, *Wing*, *VSCode*, *Spyder*, and *PyScripter* all come with the usual edit-and-run GUIs, but some add additional and advanced tools such as code refactoring and source-control integration. These IDEs also have major learning curves and are not recommended for most Python beginners. Still, you may wish to file this away for later in your Python career when you've mastered the language and move on to industrial-strength development.

Smartphone Apps

If you're running this book's examples on an Android or iOS smartphone or tablet, you'll be using an app. Some, like *Termux* on Android, come with a traditional command line and support all the REPL and file commands we've seen. In others, though, you'll launch code files with devices in the app's user interface (e.g., button taps), instead of traditional command lines. This isn't much different in spirit from running code in IDLE or other IDEs, but because details vary per app, see [Appendix A](#) for mobile platform tips, as well as your app's docs for more info.

WebAssembly for Browsers

Although an emerging technology, it's also possible to run Python code in web browsers. This is enabled today by *WebAssembly* (a.k.a. *Wasm*), which defines a portable bytecode format that is run by web browsers, much as the Python PVM runs its own bytecode (see [Chapter 2](#)). By compiling the Python interpreter's source code to this format with tools like the *Emscripten* LLVM-based compiler, web browsers are able to run Python, and hence your Python programs. While other Python-in-the-browser initiatives of the past have largely fizzled, WebAssembly is standardized by the World Wide Web Consortium (W3C) and already supported by all major desktop and mobile web browsers.

Compiling Python itself to Wasm is not trivial (and far too much to ask of Python beginners), but the *Pyodide* system has already done most of the work for you: it's a port of CPython to the WebAssembly/Emscripten platform compiled and ready to run, with JavaScript integration for access to the browser Document Object Model (DOM) and web APIs. To use Pyodide in a web page, you'll load it from a server and initialize it in a browser, with an HTML document that uses provided JavaScript code and API tools. The end result can run many Python programs in web browsers, with no local installs required.

The chief downsides of this model seem to be *speed* and *utility*. Downloading a compiled CPython interpreter to run a Python script in a browser is not quick, and the speed of Python scripts in this context may vary. Moreover, Pyodide comes with a fixed set of Python tools, and Python scripts run by browsers live in a sandbox with limited access to tools and resources on the host device. Persistent storage, for example, may support POSIX file calls and paths, but is virtual and ultimately limited in this context to options supported by browsers.

Hence, while this option may avoid some Python installs and open possibilities for Python on the web, it's not as useful for general software development as others, and its future is impossible to predict. Watch the web for more on this evolving story—including the alternative *MicroPython* for Wasm, which is smaller than CPython but implements a constrained Python subset per [Chapter 2](#), and the *py2wasm* Python-to-Wasm compiler, announced just as this book was being written.

Jupyter Notebooks for Science

We met this option earlier in conjunction with REPLs. *Jupyter* is a set of tools that allow Python code to be run in web browsers, with a focus on supporting scientific-programming tasks. Its primary and classic tools require a server to be separately installed and launched to run the code you enter in a web page, but it also comes in a form that runs code locally in browsers using the *WebAssembly* and *Pyodide* systems described in the preceding section. In both forms, Jupyter pages follow a flexible notebook paradigm, with interactive coding using the *IPython* REPL we also met earlier, code cells that run with a button click, visualization using Python numeric tools, and more.

While Jupyter is a useful and popular tool in many STEM roles, it's not targeted at general-purpose software development and isn't as broadly applicable as traditional tools like command lines stressed in this book. See other resources for usage details if Jupyter notebooks may be a part of your Python coding future.

Ahead-of-Time Compilers for Speed

Also per [Chapter 2](#), a number of AOT compiler systems, including *Nuitka* and *Shed Skin*, compile Python programs all the way to machine code, much like C and C++ compilers. Once you install such a system, you'll run its compiler on a file of Python code first, and then run the resulting program like any other executable. AOT compilers can boost program speed but add extra development steps that

slow the programming process substantially, and negate some of the advantage of using Python. Especially for Python newcomers, these systems are probably best explored after learning the Python language using more accessible options, like traditional REPLs, IDEs, and command lines.

Running Code in Code

This chapter has talked about “importing modules” a few times without really explaining what this term means. We’ll study modules and larger program architecture in depth in [Part V](#), but because imports are also a way to launch programs, this section will introduce enough module basics to get you started. Like imports, the Python `exec` built-in can be used to launch files in code too, and tools in standard-library modules let you launch programs with command lines. While we can’t go into full detail in this chapter, this section briefly surveys the launchers in this department.

Importing modules

In simple terms, every file of Python source code whose name ends in a `.py` extension is a *module*. No special code or syntax is required to make a file a module: any such file will do. Other files can access the items a module defines by *importing* that module—which essentially loads another file and grants access to that file’s contents. The contents of a module are made available to the outside world through its *attributes*—a term defined informally in the next section.

This module paradigm turns out to be the core idea behind *program architecture* in Python. Larger programs usually take the form of multiple module files, which import tools from other module files. One of the modules is designated as the main or *top-level* file, also often called the *script*—the file launched to start the entire program, which runs line by line as usual. Below this level, it’s all modules importing modules.

We’ll delve into such architectural issues in more detail later in this book. This chapter is mostly interested in the fact that import operations *run* the code in a file that is being loaded as a final step. Because of this, importing a file is yet another way to launch it. For instance, if you start an interactive session (from a system command line or otherwise), you can run the `script1.py` file we wrote earlier in [Example 3-1](#) with a simple `import` statement—which is really Python code running other Python code:

```
$ python3
>>> import script1
darwin
1267650600228229401496703205376
Hack!Hack!Hack!Hack!Hack!Hack!Hack!Hack!
```



Where to run imports: Be sure to run this Python command in the directory (i.e., folder) containing `script1.py`. Per this chapter’s earlier coverage of code folders, this is easiest if you save module files and run imports in a per-chapter folder for the chapter you’re working in. Later in this book, you’ll learn that imports in a REPL search for a module in the *current directory*, plus those listed on environment variable `PYTHONPATH` or specified otherwise. The current directory part of this will suffice for most imports you’re likely to try until then—as long as all your code files reside there.

Reloading modules

Imports work to run a file, but only once per session (really, *process*—a program run) by default. After the first import, later imports do nothing, even if you change and save the module’s source file again in another window:

```
>>> import script1
>>> import script1
```

This is by design; imports are too expensive an operation to repeat more than once per file in a given program run. As you’ll learn in [Chapter 22](#), imports must find files, compile them to bytecode, and run the code line by line, and importers usually care only that the module’s lines have defined its exports.

If you really want to force Python to run the file again in the same session without stopping and restarting the REPL, you need to instead call the `reload` function available in the `importlib` standard-library module (and previously in the now-defunct `imp` module, and a built-in function before that: that’s three incarnations, for anyone counting!):

```
>>> from importlib import reload
>>> reload(script1)
darwin
1267650600228229401496703205376
Hack!Hack!Hack!Hack!Hack!Hack!Hack!Hack!
<module 'script1' from '/Users/me/Code/script1.py'>
```

The `from` statement here simply copies a name out of a module (more on this in the next section). The `reload` function itself loads and runs the current version of your file’s source code, picking up the file’s changes if you’ve modified and saved it in another window.

This allows you to edit and use new code on the fly in the current Python interactive session. The `reload` function expects the name of an already-loaded module object, so you have to have successfully imported a module before you can reload it (and if the import failed with an error, you can’t yet reload and must import again).

Notice that `reload` also expects parentheses around the module object name, whereas `import` does not—`reload` is a function that is *called* with an argument, and `import` is a statement. That’s also why you get back an extra output line when reloading—the odd last line is just the display representation of the `reload` call’s return value, a Python module object. You’ll learn more about using functions in general in [Chapter 16](#); for now, when you hear “function,” remember that parentheses are required to run a call, even if there’s nothing to send.

Module attributes: a first look

Imports and reloads provide a natural program launch option because import operations execute files as a last step. In the broader scheme of things, though, modules serve the role of *libraries* of tools, as you’ll learn in [Part V](#). The basic idea is straightforward, though: a module is mostly just a package of variable names, known as a *namespace*, and the names within that package are called *attributes*—variable names that are attached to a specific object (like a module).

In more concrete terms, all the names assigned at the top level of a module’s file become its attributes, and a module’s importers gain access to all of them. These names are usually assigned to tools exported by the module—functions, classes, variables, and so on—that are intended to be used in other files and other programs. Externally, a module file’s names can be fetched with two Python statements, `import` and `from`, and may be reset with the `reload` call.

To illustrate, use a text editor to create a one-line Python module file called *myfile.py* in your working directory, with the contents in [Example 3-2](#).

Example 3-2. *myfile.py*

```
title = 'Learning Python, 6th Edition'
```

This may be one of the world's simplest Python modules (it contains a single assignment statement), but it's enough to illustrate the point. When this file is imported, its code is run to generate the module's attribute. That is, the assignment statement creates a variable and module attribute named `title`.

You can access this module's `title` attribute in other components in two different ways. First, you can load the module as a whole with an `import` statement, and then *qualify* the module name with the attribute name to fetch it (note that we're letting the interpreter print automatically here):

```
$ python3                                # Start Python REPL
>>> import myfile                        # Run file, load module as a whole
>>> myfile.title                         # Use its attribute names: '.' to qualify
'Learning Python, 6th Edition'
```

In general, the dot expression syntax *object.attribute* lets you fetch any attribute attached to any object and is one of the most common operations in Python code. Here, we've used it to access the string variable `title` inside the module `myfile`—in other words, `myfile.title`.

Alternatively, you can fetch (really, copy) names out of a module with `from` statements:

```
$ python3                                # Restart Python REPL
>>> from myfile import title             # Run file, copy its names
>>> title                               # Use name directly: no need to qualify
'Learning Python, 6th Edition'
```

As you'll see in more detail later, `from` is just like an `import`, with an extra assignment to names in the importing code. Technically, `from` copies a module's *attributes*, such that they become simple *variables* in the recipient. So, you can refer to the imported string this time as `title` (a variable) instead of `myfile.title` (an attribute).

Naturally, modules usually define more than one name to be used both in and outside their files. [Example 3-3](#), for instance, defines three.

Example 3-3. *threenames.py*

```
a = 'PC'                                # Define three attributes
b = 'Phone'                             # Exported to other files
c = 'Tablet'
print(a, b, c)                           # Also used as variables in this file
```

This code file, *threenames.py*, assigns three variables, and so generates three attributes for the outside world. It also uses its own three variables in a `print` statement, as you see when this is run as a top-level file from a system prompt:

```
$ python3 threenames.py
PC Phone Tablet
```

All of this file's code also runs as usual the first time it is imported elsewhere, by either an `import` or `from`. Clients of this file that use `import` get a module with attributes, while clients that use `from` get copies of the file's names:

```

$ python3
>>> import threenames                # Grab the whole module: it runs here
PC Phone Tablet

>>> threenames.b, threenames.c        # Access its attributes: qualify to use
('Phone', 'Tablet')

>>> from threenames import b, c        # Copy multiple names out: use directly
>>> b, c
('Phone', 'Tablet')

```

The results here are printed in parentheses because they are really *tuples*—a kind of object created by the comma in the inputs (and covered in the next part of this book)—that you can safely accept on faith for now.

From a grander perspective, modules form the highest layer of Python program architecture: as self-contained namespaces, they naturally support code organization and reuse, and they automatically minimize name collisions in your code. We'll deal with their loftier goals later in this book.

For this chapter, it's enough to know that imports and reloads are another way to run your code files, though probably a *secondary option*: due to complicated quirks that we'll skip here (e.g., `reload` updates importers using `import` but not `from`), tools like command lines and IDEs are generally better bets for running Python code.

The `exec` built-in

Python also provides a way to launch files with code that does not rely on the module concepts of the preceding section. The `exec` built-in function compiles and runs whatever Python source code statements are in the string you pass to it. Along with its `eval` expression cousin, this supports many dynamic roles you'll meet in upcoming chapters.

By passing a code file's loaded contents to `exec`, though, this yields another way to launch code files from a REPL or other file without having to import and later reload. Each such `exec` runs the *current* version of the code read from a file, without requiring imports or reloads. For instance, using

Example 3-1's *script1.py* again:

```

$ python3
>>> exec(open('script1.py').read())
darwin
1267650600228229401496703205376
Hack!Hack!Hack!Hack!Hack!Hack!Hack!Hack!

```

To understand this code, you must first know that the `open` and `read` nested inside it run first, and left to right, and load the file's entire contents as a string. We'll study files in the next part of this book, so take this as a preview for now. (This was also a forward knowledge dependency added in Python 3.X, but that's what this book presents.)

Once the file is loaded, though, the `exec` call has an effect similar to an import, but it doesn't actually import the module. Instead, each time you run an `exec/open` combo this way it runs the file's code anew and unconditionally—as though you had pasted the file's code at the place where `exec` is called. As one consequence, this `exec/open` scheme does not require module reloads after file changes.

On the downside, because it works as if you've pasted code into the place where it is called, `exec` has the potential to silently overwrite variables you may currently be using. For example, our *script1.py* assigns to a variable named `x`. If that name is also being used in the place where `exec` is called, the name's value is replaced—sans warning:

```
>>> x = 999
>>> exec(open('script1.py').read())    # Code run in this namespace by default
...same output...
>>> x                                  # Its assignments can overwrite names here
'Hack!'
```

This potential for name collisions is a downside shared by the last section's `from` statement. By contrast, the basic `import` statement makes the file a separate module namespace so that its assignments will not change variables in the importer's scope. The price you pay for its namespace partitioning of modules is the need to reload and qualify.

Command-line launchers

Finally, Python code can also be run by Python code that uses standard-library tools that spawn command lines in parallel processes. The `os` module's `system` call, for instance, runs a command as if you typed it at a system console (its `0` return value echoed at the end here means all worked well):

```
$ python3
>>> import os
>>> os.system('python3 script1.py')
darwin
126765060022829401496703205376
Hack!Hack!Hack!Hack!Hack!Hack!Hack!Hack!
0
```

This `os` module's `popen` call does similar but returns a file object from which you can read the spawned command's printed output as a string for use in the spawning code; and Python's `subprocess.run` call can be used to launch programs by command lines with much more control over the fine points. We'll deploy `popen` in [Chapter 21](#).

Though related to running code, all these tools are well beyond the scope of this getting-started chapter, so consult Python's library manuals for the full story if and when such utilities become useful in your future work. And a caution: these tools, along with `exec`, will happily run any command line you throw at them—including *one to erase all your files!*—so they're best limited to running Python commands unless and until you're sure that other commands are safe.

Other Launch Options

You've already seen a blizzard of code-launching options for Python, but this accounting is still incomplete. For example, Python code can also be run today by:

- Programs written in C, C++, Java, and more, which *embed* Python and Python code
- Text *editors* that aren't full IDEs, but know how to run Python code you're editing
- Excel *spreadsheets*, when calculating a sheet's cells coded in Python
- Web *servers* that spawn scripts automatically in response to browser UI actions
- Users launching *standalone executables*, which we explored in [Chapter 2](#)
- And (as usual) more

Moreover, launching techniques tend to evolve as rapidly as everything else in computing, and future options are impossible to foresee. In general, because Python keeps pace with such changes, you should be able to launch Python programs in whatever way makes sense for the machines you use, both now

and in the future—be that by swiping on your smartphone, grabbing icons in a virtual reality, or shouting a script's name over your coworkers' conversations.

Which Option Should I Use?

With all these options, beginners might naturally ask, Which one is best for me? In general, you should try both basic command lines and the IDLE GUI if you are just getting started with Python, unless you'll be working in smartphone apps or web-based notebooks with interfaces that are more unique. Command lines are simple and powerful, but IDLE provides a user-friendly GUI environment that hides some underlying details and works the same on all PCs.

If, on the other hand, you are an experienced programmer, you might be more comfortable with simply the text editor of your choice in one window, and a system console interface in another for launching edited programs via Python command lines. Because the choice of development environments is very subjective, this book can't offer much more in the way of universal guidelines. In general, whatever environment you like to use will be the best for you to use.

Chapter Summary

In this chapter, we've explored common ways to launch Python programs: by running code typed interactively and by running code stored in files with system command lines, file icon clicks, IDE GUIs such as IDLE, and more. We've covered a lot of pragmatic startup territory here. This chapter's goal was to equip you with enough information to enable you to start writing some code, which you'll do in the next part of this book. There, we will start exploring the Python language itself, beginning with its core *data types*—the objects that are the subjects of your programs.

First, though, take the usual chapter quiz to exercise and review what you've learned here. Because this is the last chapter in this part of the book, it's followed with a set of more complete exercises that test your mastery of this entire part's topics. For help and answers for the latter set of problems, or just for a refresher, be sure to turn to [Appendix B](#) after you've given the exercises a try.

Test Your Knowledge: Quiz

1. How can you start a Python interactive interpreter session (REPL)?
2. Where do you type a system command line to launch a script file?
3. Name four or more ways to run the code saved in a script file.
4. What pitfall is related to clicking file icons on Windows and Linux?
5. Why might you need to reload a module?
6. How do you run a script from within the IDLE GUI?
7. How are modules, attributes, and namespaces related?

Test Your Knowledge: Answers

1. A Python interactive session can be started by typing a Python command line in your system's console window. Type `py` on Windows and `python3` everywhere else and type it into a *Command Prompt* or *PowerShell* window on Windows, *Terminal* on macOS and Linux, and the *Termux* app on Android. Another alternative is to launch *IDLE*, as its main Shell window is an interactive session. Other IDEs, smartphone apps, and browser-based systems may offer REPLs in more unique ways.
2. You type system command lines in the same interface used to launch an interactive session by command line. This is whatever your platform provides as a system console: again, *Command Prompt* or *PowerShell* on Windows, *Terminal* on macOS and Linux, *Termux* on Android, or other. You type this at the system's prompt shown as `$` in this book, not at the Python interactive interpreter's `>>>` prompt used to enter Python code—be careful not to confuse these prompts, because Python and the system shell are different systems.
3. Code in a script (really, module) file can be run with system command lines, file icon clicks, imports and reloads, the `exec` built-in function, `os` module tools, and IDE GUI devices such as *IDLE*'s *Run*→*Run Module* menu option. Some platforms support more specialized launching techniques, like drag and drop on macOS, app UIs on smartphones, and web notebooks. In addition, some text editors have unique ways to run Python code, some Python programs are provided and run as standalone executables, and some systems use Python code in embedded mode, where it is run by an enclosing program written in another language. Though in its early days, code may also be run in web browsers with a Python port to WebAssembly like *Pyodide*.
4. Scripts that print and then exit cause the output file to disappear immediately, before you can view the output. `input()` can pause before exit so the output window stays open, but error messages generated by your script also appear in an output window that closes before you can examine its contents—and before an `input()` pause is reached. Hence, system command lines and IDEs such as *IDLE* are better for most development.
5. Python imports (i.e., loads) a module only once per process, by default, so if you've changed its source code and want to run the new version without stopping and restarting Python, you'll have to reload it. You must import a module at least once before you can reload it. Running files of code from a system shell command line, via an icon click, or in an IDE such as *IDLE* generally makes this a nonissue, as those launch schemes usually run the current version of the source code file each time. An `exec/open` pair can avoid reloads too.
6. Within the text edit window of the file you wish to run, select the window's *Run*→*Run Module* menu option. This runs the window's source code as a top-level script file in *IDLE* and displays its output back in the interactive Python "Shell" window.
7. Each module file is automatically a namespace—that is, a package of variables reflecting the assignments made at the top level of the file. Each of the module's variables becomes an attribute of the module when it's imported and are accessed by `"."` qualification or `from` name copies. Namespaces help avoid name collisions in Python programs: because each module file is a self-contained namespace, files must explicitly import other files in order to use their names.

Test Your Knowledge: Part I Exercises

It's time to start doing a little coding on your own. This first exercise session is fairly simple, but it is designed to make sure you're ready to work along with the rest of the book, and a few of its questions hint at topics to come in later chapters. Be sure to check the section “[Part I, Getting Started](#)” on page 1157 in [Appendix B](#) for the answers; the exercises and their solutions sometimes contain supplemental information not discussed in the main text, so you're encouraged to take a peek at the solutions even if you manage to answer all the questions on your own.

1. *Interaction:* Using a system command line, IDLE, or any other method that works on your device, start the Python interactive command line (the `>>>` prompt, a.k.a. REPL) and type the expression `'Hello World!'` (including the quotes). The string should be echoed back to you. The purpose of this exercise is to get your environment configured to run Python. In rare scenarios, you may need to type the full path to the Python executable or add its path to your `PATH` environment variable. See [Appendix A](#) for tips on environment-variable settings if needed.
2. *Programs:* With the text editor of your choice, write a simple module file containing the single statement `print('Hello module world!')` and store it as `module1.py`. Now, run this file by using any launch option you like: running it in IDLE, clicking or tapping on its file icon, starting it by command line in a console (e.g., `python3 module1.py`), executing it in code with tools like `exec` and `imports/reloads`, or by using UI options in apps, other IDEs, and web notebooks. In fact, experiment by running your file with as many of the launch techniques discussed in this chapter as you can. Which technique seems easiest to you?
3. *Modules:* Start the Python interactive command line (`>>>` prompt) and import the module you wrote in exercise 2. Try moving the file to a different directory and importing it again from its original directory (i.e., run Python in the original directory when you import). What happens? Hint: there's still a `.pyc` bytecode file for `module1` in a `__pycache__` subdirectory there, but it's named oddly. In general, imports search for modules in the current directory, plus every directory listed on the `PYTHONPATH` environment variable—as you'll learn in [Part V](#).
4. *Scripts:* If your platform supports it, copy your `module1.py` module file to another named `script1.py`; then add the `#!` line to the top of `script1.py`, give this file executable privileges, and run it directly as an executable (e.g., sans the “python3”). What does the first line need to contain? `#!` lines traditionally have meaning only on Unix-like platforms (e.g., macOS, Linux, and Android), but also apply to Windows today, thanks to the py Windows launcher. If you're working on Windows, also try running your file by listing just its name in a Command Prompt window without the word `py` before it, via the Start menu, Windows+R Run dialog, or other schemes. On macOS, try a drag-and-drop to the Python Launcher app in Applications (or elsewhere).
5. *Errors and debugging:* Experiment with typing mathematical expressions and assignments at the Python interactive command line (that is, REPL). Along the way, type the expressions `2 ** 500` and `1 / 0`, and reference an undefined variable name as we did early on in this chapter. What happens?

You may not know it yet, but when you make a mistake, you're doing exception processing, a topic we'll explore in depth in [Part VII](#). As you'll learn there, you are technically triggering what's known as the *default exception handler*—logic that prints a standard error message. If you do not catch an error, the default handler does and prints the standard error message in response.

Exceptions are also bound up with the notion of *debugging* in Python. When you're first starting out, Python's default error messages on exceptions will probably provide as much error-handling support as you need—they give the cause of the error, as well as showing the lines in your code that were active when the error occurred. For more about debugging, see the upcoming sidebar “Debugging Python Code” on page 49.

6. *Breaks and cycles*: In any Python REPL, type:

```
L = [1, 2]           # Make a 2-item list
L.append(L)          # Append L as a single item to itself
L                    # Print L: a cyclic/circular object
```

What happens? In all but the most ancient of Python versions, you'll see a strange output that is described in the [Appendix B](#) solution, and which will make more sense when you study object *references* in the next part of the book. Why do you think your version of Python responds the way it does for this code?

7. *Documentation*: Spend at least five minutes browsing the Python library and language manuals before moving on to get a feel for the available tools in the standard library and the structure of the documentation set. It takes at least this long to become familiar with the locations of major topics in the manual set; once you've done this, it's easy to find what you need. You can find this manual via the Python Start menu entry on some Windows, in the Python Docs option on the Help pull-down menu in IDLE, or online at <http://www.python.org/doc>. You'll also learn more about the manuals and other documentation sources (including PyDoc and the `help` function) in [Chapter 15](#). If you still have time, go explore the Python website, as well as its PyPI third-party extension repository; python.org's About and Search, for example, may be useful when you're getting started.

Debugging Python Code

Naturally, none of this book's readers ever have bugs in their code (insert smiley here), but for less fortunate friends of yours who may, here's a quick review of the strategies commonly used by real-world Python programmers to debug errors in their code. The first two may be all you'll need early in the learning process, but others may grow important when you start writing larger scripts, and all are useful to review now before you start coding in earnest in the next chapter—and making the mistakes that are inevitable in programming:

- **Do nothing.** This doesn't mean that Python programmers don't debug their code. But when you make a mistake in a Python program, you get a useful and readable error message. The message pinpoints the location of the error in your code by file and line and may even offer a suggested “Did you...?” fix as you saw earlier in this chapter. If you already know Python, and especially for your own code, this is often enough—read the error message and fix the tagged line. In many cases, this *is* debugging in Python.
- **Insert `print` statements.** Probably the main way that Python programmers debug their code is to insert `print` statements and run again. Because most Python code runs immediately after changes, this is usually the quickest way to get more information than error messages provide. The `print` statements don't have to be sophisticated—a simple “I am here” beacon or display of variable values is usually enough to provide the context you need. If you write programs

that don't have a console window (e.g., some GUIs and apps), you may need to find your printed messages in an automatic logfile, or use the next point.

- **Insert calls to the logging module.** The beacon messages of the prior point may also use Python's logging module instead of `print`, to make the process more formal and gain better control of the output. Because this requires some planning, it's more common in larger programs than tactical scripts. See Python's library manual for logging usage details.
- **Use GUI debuggers.** For larger systems you didn't write, and for beginners who want to trace code in more detail, most Python development GUIs have some sort of point-and-click debugging support. IDLE has a debugger too, but it doesn't seem to be used very often in practice—perhaps because it has no command line, or simply because adding `print` statements is often quicker than setting up a debugging session. To learn more about the debugger, see IDLE's Help menu, or simply try it on your own; its basic interface is described earlier in this chapter. For similar debugging support in other IDEs, see their docs.
- **Use the pdb command-line debugger.** For full control, Python comes with a source code debugger named *pdb*, available as a module in Python's standard library. With *pdb*, you type commands to step line by line, display variables, set and clear breakpoints, continue to a breakpoint or error, and so on. You can launch *pdb* by either importing it and calling `pdb.run('code')` or running it as a top-level script with the command `python3 -m pdb file.py`. You can also import and call *pdb*'s postmortem `pdb.pm()` after an error occurs to get more information about what went wrong. We'll revisit *pdb* in [Chapter 36](#), but see Python's library manual and *pdb*'s own `help` command for more usage tips.
- **Use Python's -i command-line argument.** Short of adding prints or running debuggers, you can still see what went wrong on errors. If you run your script from a command line and pass a `-i` argument between Python and the name of your script (e.g., `python3 -i script.py`), Python will automatically open its interactive mode (the `>>>` prompt) when your script exits, whether it ends successfully or runs into an error. You can then print the final values of variables to get more details about what happened in your code, or import and run *pdb*'s debugger or postmortem mode.
- **Catch and handle errors in code.** Ultimately, errors are a well-defined mechanism in Python known as *exceptions*, which you can catch, process, and recover from in your own code. You'll learn how in [Part VII](#).

Perhaps the best takeaway on debugging is that errors are detected and reported in Python as a norm, rather than passing silently or crashing the system altogether. Making mistakes is never fun, of course, but especially for those who recall when debugging meant getting out a hex calculator and poring over piles of memory-dump printouts, Python's debugging support makes errors a lot less painful than they might otherwise be.

Objects and Operations

Introducing Python Objects

This chapter begins our tour of the Python language. In an informal sense, in Python we do *things with stuff*. “Things” take the form of operations like addition and concatenation, and “stuff” refers to the objects on which we perform those operations. In this part of the book, our focus is on that *stuff*, and the *things* our programs can do with it.

Somewhat more formally, in Python, data takes the form of *objects*—either built-in objects that Python provides, such as strings and lists, or add-on objects we create with Python classes or external-language tools. As you’ll find, these objects are essentially just pieces of memory, with values and associated operations. Moreover, *everything* is an object in a Python script. Even simple numbers qualify, with values (e.g., 99) and supported operations (+, -, and so on).

Because objects are also the most fundamental notion in Python programming, this chapter gets us started with a survey that previews Python’s built-in object types. Later chapters in this part provide a second pass that fills in details we’ll gloss over in this survey. Here, our goal is a brief tour to introduce the basics.

The Python Conceptual Hierarchy

Before we get to the code, let’s first establish a clear picture of how this chapter fits into the overall Python picture. From a more concrete perspective, Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

The discussion of modules in [Chapter 3](#) introduced the highest level of this hierarchy. This part’s chapters begin at the bottom—exploring both built-in objects and the expressions you can code to use them.

We’ll move on to statements in the next part of the book, though you will find that they largely exist to manage the objects you’ll meet here. Furthermore, by the time we reach classes in the OOP part of this book, you’ll discover that they allow you to define new object types of your own, by both using and

emulating the object types you will explore here. Because of all this, built-in objects are a mandatory point of embarkation for all Python journeys.



Terminology moment: Traditional introductions to programming often stress its three pillars of *sequence* (“Do this, then that”), *selection* (“Do this if that is true”), and *repetition* (“Do this many times”). Python has tools in all three categories, and these terms might help you organize your thinking early on. But they are also artificial and simplistic, and prone to confuse. For example, tools such as comprehensions are both repetition and selection; these terms have other, more specific meanings in Python; and many later concepts won’t seem to fit this mold at all. In Python, the more strongly unifying principle is *objects* and what we can do with them. To see why, read on.

Why Use Built-in Objects?

If you’ve used lower-level programming languages, you know that much of your work centers on implementing *objects*—also known as *data structures*—to represent the components in your application’s domain. You may need to lay out memory structures, manage memory allocation, implement search and access routines, and so on. These chores are about as tedious (and error-prone) as they sound, and they usually distract from your program’s real goals.

In typical Python programs, most of this grunt work goes away. Because Python provides powerful object types as an intrinsic part of the language, there’s usually no need to code object implementations before you start solving problems. In fact, unless you have a need for special processing that built-in objects don’t provide, you’re almost always better off using a built-in object instead of implementing your own. Here are some reasons why:

- **Built-in objects make programs easy to write.** For simpler tasks, built-in objects are often all you need to represent the structure of problem domains. Because you get powerful tools such as collections (lists) and search tables (dictionaries) for free, you can use them immediately. You can get a lot of work done with Python’s built-in object types alone.
- **Built-in objects are components of extensions.** For more complex tasks, you may need to provide your own objects using Python classes or C-language interfaces. But as you’ll see in later parts of this book, objects implemented manually are often built on top of built-in objects such as lists and dictionaries. For instance, a stack data structure may be implemented as a class that manages or customizes a built-in list.
- **Built-in objects are often more efficient than custom data structures.** Python’s built-in objects employ algorithms that have already been optimized and are often implemented in a lower-level language like C for speed. Although you can write similar object types on your own, you’ll usually be hard-pressed to match the level of performance that built-in object types provide.
- **Built-in objects are a standard part of the language.** In some ways, Python borrows both from languages that rely on built-in tools (e.g., Lisp) and languages that rely on the programmer to provide tool implementations of their own (e.g., C++). Although you can implement unique object types in Python, you don’t need to do so just to get started. Moreover, because Python’s built-ins are standard, they’re always the same; proprietary toolkits, on the other hand, tend to differ from site to site.

In other words, not only do built-in object types make programming easier, they’re also more powerful and accessible than most of what can be created from scratch. Regardless of whether you implement new object types, built-in objects form the core of every Python program.

Python’s Core Object Types

Table 4-1 previews Python’s built-in objects, and some of the syntax used to code their *literals*—that is, the expressions that generate these objects. Some of these objects will probably seem familiar if you’ve used other languages; for instance, numbers and strings represent numeric and textual values, respectively, and file objects provide an interface for processing real files stored on your computer.

To some readers, though, the object types in **Table 4-1** may be more general and flexible than what you are accustomed to. For instance, you’ll find that lists and dictionaries alone are powerful data representation tools that obviate most of the work you do to support collections and searching in lower-level languages. In short, lists provide ordered collections of other objects, while dictionaries store objects by key, and both come with automatic memory management, support arbitrarily nesting, can grow and shrink on demand, and may contain objects of any kind.

Table 4-1. Python built-in (core) objects

Object type	Example literals/creation
Numbers	1234, 3.1415, 0b111, 1_234, 3+4j, Decimal, Fraction
Strings	'code', "app's", b'a\x01c', 'h\u00c4ck', 'hÄck🐍'
Lists	[1, [2, 'three'], 4.5], list(range(10))
Dictionaries	{'job': 'dev', 'years': 40}, dict(hours=10)
Tuples	(1, 'app', 4, 'U'), tuple('hack'), namedtuple
Files	open('docs.txt'), open(r'C:\data.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core objects	Booleans, types, None
Program-unit objects	Functions, modules, classes (Parts IV, V, and VI)
Implementation objects	Compiled code, stack tracebacks (Parts IV and VII)

Also shown in **Table 4-1**, *program units* such as functions, modules, and classes—which you’ll meet in later parts of this book—are objects in Python too; they are created with statements and expressions such as `def`, `class`, `import`, and `lambda` and may be passed around scripts freely, stored within other objects, and so on. Python also provides a set of *implementation-related* objects such as compiled-code objects, which are generally of interest to tool builders more than application developers; we’ll explore these later, though in less depth due to their specialized roles.

Despite its title, **Table 4-1** isn’t really complete because *everything* we process in Python programs is a kind of object. For instance, when we perform text pattern matching in Python, we create pattern objects, and when we do network scripting, we use socket objects. These other kinds of objects are generally created by importing and using functions in standard or add-on library modules, and have behavior all their own. Patterns and sockets, for example, are made by calling tools in the standard library’s `re` and `socket` modules, respectively.

We usually call the objects in **Table 4-1** *core object types*, though, because they are effectively built into the Python language itself—that is, there is specific expression syntax for generating most of them. For

instance, when you run the following code with characters surrounded by quotes in a REPL or program file:

```
>>> 'Python'
```

you are, technically speaking, running a *literal expression* that generates and returns a new *string* object. There is Python language syntax to make this object. Similarly, an expression wrapped in square brackets makes a *list*, one in curly braces makes a *dictionary* or *set*, and so on. Even though, as you'll see, Python does not require or use type declarations, the syntax of the expressions you run determines the types of objects you create and use. In fact, object-generation expressions like those in [Table 4-1](#) are generally where types originate in the Python language.

Just as importantly, once you create an object, you bind its operation set for all time—you can perform only string operations on a string and list operations on a list. In formal terms, this means that Python is *dynamically typed*, a model that keeps track of object types for you automatically instead of requiring declaration code, but it is also *strongly typed*, a constraint that means you can perform on an object only operations that are valid for its type.

We'll study each of the object types in [Table 4-1](#) completely in upcoming chapters. Before digging into the full details, though, let's begin by taking a quicker look at Python's core objects in action. The rest of this chapter provides a preview of the operations we'll explore in more depth in the chapters that follow. Don't expect to find the full story here—the goal of this chapter is just to whet your appetite and introduce some key ideas. Still, the best way to get started is to get started, so let's jump right into some real object-wrangling code.

Numbers

If you've done any programming or scripting in the past, some of the object types in [Table 4-1](#) will probably seem familiar. Even if you haven't, numbers are fairly straightforward. Python's core objects set includes the usual suspects: *integers* that have no fractional part, *floating-point* numbers that do, and more exotic types—*complex* numbers with imaginary parts, *decimals* with flexible precision, *rational*s with numerator and denominator, and full-featured *sets*. Built-in numbers are enough to represent most numeric quantities—from your age to your bank balance—but specialized numeric objects like vectors and matrixes are also available as third-party add-ons.

Although they offer fancier options, Python's basic number objects are, well, basic. Numbers in Python support the normal mathematical operations. For instance, the plus sign (+) performs addition, a star (*) is used for multiplication, and two stars (**) are used for exponentiation. Here is a demo in a Python REPL of the sort we learned about in [Chapter 3](#):

```
$ python3                                # Start up a REPL
>>> 123 + 222                            # Integer addition
345
>>> 1.5 * 4                              # Floating-point multiplication
6.0
>>> 1_234_567, 0x15, bin(21)             # Separators, hex, binary
(1234567, 21, '0b10101')
>>> 2 ** 100                             # 2 to the power 100, again
1267650600228229401496703205376
```

Notice the last result here: Python's integer object automatically provides extra precision for large numbers like this when needed. You can, for instance, compute 2 to the power 12,345 as an integer in Python, but you probably shouldn't try to grok the result—with nearly 4K digits, it's a lot to take in:

```
>>> len(str(2 ** 12345))      # How many digits in a really BIG number
3717
```

This nested-call form works *from inside out*—first computing the `**` result, then converting it to a string of digits with the built-in `str` function, and finally getting the length of the resulting string with `len`. The end result is the number of digits in the number. `str` and `len` both work on many object types, and we'll use them again as we move along.

Besides expressions, there are a handful of useful numeric modules that ship with Python in *modules*—which are just packages of additional tools that we import to use, using statements like `import` introduced in [Chapter 3](#):

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
```

The `math` module contains more advanced numeric tools as functions, while the `random` module performs random-number generation and random selections (here, from a Python *list* coded in square brackets—an ordered collection of other objects to be introduced later in this chapter):

```
>>> import random
>>> random.random()
0.7082048489415967
>>> random.choice([1, 2, 3, 4])
2
```

Python also includes more exotic numeric objects—such as complex, fixed-precision, and rational numbers, as well as sets and Booleans—and the third-party open source extension domain has even more (e.g., matrixes and vectors, and extended precision numbers). We'll defer discussion of these objects until later in this chapter and book.

So far, we've been using Python much like a simple calculator; to do better justice to its built-in objects catalog, let's move on to explore strings.



Big numbers versus DOS attacks: Python integers can be arbitrarily large and may even be used to represent floating-point values with extended precision. While integer size is limited only by your computer's memory, though, Python 3.11 and later require extra steps to convert pathologically large numbers to decimal strings (e.g., for `str` or display). This avoids rare denial-of-service attacks with a 4,300-digit limit that can be lifted with a `sys` module tool, `set_int_max_str_digits`; call this to count digits in larger numbers, and see Python's docs for all the sordid details.

Strings

Strings are used to record both textual information (your name, for instance) as well as arbitrary collections of bytes (such as an image file's contents). They are our first example of what in Python we call a *sequence*—a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative positions. Strictly speaking, strings are sequences of one-character strings; other, more general sequence objects include *lists* and *tuples*, covered later.

Sequence Operations

As sequences, strings support operations that assume a positional ordering among items. For example, if we have a four-character string coded inside *quotes* (of the single or double straight kind—they mean the same thing, but single is more common and less busy), we can verify its length with the built-in `len` function and fetch its components with *indexing* expressions:

```
>>> S = 'Code'           # Make a 4-character string, and assign it to a name
>>> len(S)               # Length: number characters
4
>>> S[0]                 # The first item in S, indexing by zero-based position
'C'
>>> S[1]                 # The second item from the left
'o'
```

In Python, indexes are coded in square brackets as offsets from the front, so start from 0: the first item is at index 0, the second is at index 1, and so on.

Notice how we assign the string to a *variable* named `S` here. We'll go into detail on how this works later (especially in [Chapter 6](#)), but Python variables never need to be declared ahead of time. A variable is created when you assign it a value, may be assigned any type of object, and is replaced with its value when it shows up in an expression. It must also have been previously assigned by the time you use its value. For the purposes of this chapter, it's enough to know that we need to assign an object to a variable in order to save it for later use.

In Python, we can also index backward, from the *end*—positive indexes count forward from the left, and negative indexes count backward from the right. Continuing our REPL session:

```
>>> S[-1]               # The last item from the end in S
'e'
>>> S[-2]               # The second-to-last item from the end
'd'
```

Formally, a negative index is simply added to the string's length to yield a positive offset, so the following two operations are equivalent (though the first is easier to code and less easy to get wrong):

```
>>> S[-1]               # The last item in S
'e'
>>> S[len(S)-1]         # Negative indexing, the hard way
'e'
```

Notice that we can use an *arbitrary expression* in the square brackets, not just a hardcoded number literal—anywhere that Python expects a value, we can use a literal, a variable, or any expression we wish. Python's syntax is completely general this way.

In addition to simple positional indexing, sequences also support a more general form of indexing known as *slicing*, which is a way to extract an entire section (a.k.a. slice) in a single step. For example:

```
>>> S                   # A 4-character string
'Code'
>>> S[1:3]              # Slice of S from offsets 1 through 2 (not 3)
'od'
```

Probably the easiest way to think of slices is that they are a way to extract an entire *column* from a string in a single step. Their general form, `X[I:J]`, means “give me everything in *X* from offset *I* up to but not including offset *J*.” The result is returned in a new object. The second of the preceding operations, for

instance, gives us all the characters in string `S` from offsets 1 through 2 (that is, 1 through 3 – 1) as a new string. The effect is to slice or “parse out” the two characters in the middle at offsets 1 and 2.

In a slice, the left bound defaults to zero, and the right bound defaults to the length of the sequence being sliced. This leads to some common usage variations:

```
>>> S[1:]                # Everything past the first (1:len(S))
'ode'
>>> S                    # S itself hasn't changed
'Code'
>>> S[0:3]               # Everything but the last
'Cod'
>>> S[:3]                # Same as S[0:3]
'Cod'
>>> S[:-1]               # Everything but the last again, but simpler (0:-1)
'Cod'
>>> S[:]                 # All of S as a top-level copy (0:len(S))
'Code'
```

Note in the second-to-last command how negative offsets can be used to give bounds for slices, too, and how the last operation effectively copies the entire string. As you’ll learn later, there is no reason to copy a string, but this form can be useful for other sequences like lists.

Finally, as sequences, strings also support *concatenation* with the plus sign (joining two strings into a new string) and *repetition* (making a new string by repeating another):

```
>>> S
'Code'
>>> S + 'xyz'            # Concatenation
'Codexyz'
>>> S                    # S is unchanged
'Code'
>>> S * 8                # Repetition
'CodeCodeCodeCodeCodeCodeCodeCode'
```

Notice that the plus sign (+) means different things for different objects: addition for numbers, and concatenation for strings. This is a general property of Python that we’ll regularly call *polymorphism* in this book—in short, this means that the meaning of an operation depends on the objects being operated on.

As you’ll see when we study dynamic typing, this polymorphism property accounts for much of the conciseness and flexibility of Python code. Because object types aren’t constrained, a Python-coded operation can normally work on many different types of objects automatically, as long as they support a compatible interface (like the + operation here). This turns out to be a huge idea in Python; you’ll learn more about it later on this tour.

Immutability

Also notice in the prior examples that we were not changing the original string with any of the operations we ran on it. As it turns out, every string operation is defined to produce a new string as its result, because strings are *immutable* in Python—they cannot be changed in place after they are created.

More generally, you can never overwrite the values of immutable objects. For example, you can’t change a string by assigning to one of its positions, but you can always build a new one and assign it to the same name. Because Python automatically cleans up old objects as you go (as you’ll learn later), this isn’t as inefficient as it may sound:

```
>>> S
'Code'

>>> S[0] = 'z'           # Immutable objects cannot be changed
...error text omitted...
TypeError: 'str' object does not support item assignment

>>> S = 'Z' + S[1:]      # But we can run expressions to make new objects
>>> S
'Zode'
```

Every object in Python is classified as either immutable (unchangeable) or not. In terms of the core objects, *numbers*, *strings*, and *tuples* are immutable; but *lists*, *dictionaries*, and *sets* are not—they can be changed in place freely, as can most new objects you’ll code with classes. This distinction turns out to be crucial in Python work, in ways that we can’t yet fully explore. Among other things, immutability can be used to guarantee that an object remains constant throughout your program; mutable objects’ values, by contrast, can be changed at any time and place (and whether your code expects it or not!).

Strictly speaking, you can change text-based data *in place* if you either expand it into a *list* of individual characters and join it back together with nothing between, or use the special-purpose `bytearray` object:

```
>>> S = 'Python'
>>> L = list(S)           # Expand to a list: [...]
>>> L
['P', 'y', 't', 'h', 'o', 'n']
>>> L[0] = 'C'           # Change it in place
>>> ''.join(L)           # Join with empty delimiter
'Cython'

>>> B = bytearray(b'app') # A bytes/list hybrid (ahead)
>>> B.extend(b'lication') # 'b' means bytes string
>>> B
bytearray(b'application') # B[i] = ord(x) works here too
>>> B.decode()           # Translate to normal string
'application'
```

The `bytearray` supports in-place changes for text, but only for text whose characters are all at most 8-bits wide (e.g., ASCII). All other strings are still immutable—`bytearray` is a distinct hybrid of immutable *bytes* strings (whose `b'...'` syntax distinguishes them from normal text strings) and mutable *lists* (coded and displayed in `[]`), but we have to wait until we learn more about both these and Unicode text ahead to fully grasp this code.

Type-Specific Methods

Every string operation we’ve studied so far is really a sequence operation—that is, these operations will work on other sequences in Python as well, including lists and tuples. In addition to generic sequence operations, though, strings also have operations all their own, available as *methods*—functions that are attached to and act upon a specific object, which are triggered with a call expression using parentheses.

For example, the string `find` method is the basic substring search operation (it returns the offset of the passed-in substring, or `-1` if it is not present), and the string `replace` method performs global searches and replacements; both act on the subject that they are attached to and called from:

```

>>> S = 'Code'
>>> S.find('od')           # Find the offset of a substring in S
1
>>> S
'Code'
>>> S.replace('od', 'abl')  # Replace all substrings 'od' in S with 'abl'
'Cable'
>>> S
'Code'

```

Again, despite the names of these string methods, we are not changing the original strings here but creating new strings as the results—because strings are immutable, this is the only way this can work. String methods are the first line of text-processing tools in Python. Other methods split a string into substrings on a delimiter (handy as a simple form of parsing), perform case conversions, test the content of the string (digits, letters, and so on), and strip whitespace characters off the ends of the string:

```

>>> line = 'aaa,bbb,ccccc,dd'
>>> line.split(',')         # Split on a delimiter into a list of substrings
['aaa', 'bbb', 'ccccc', 'dd']

>>> S = 'code'
>>> S.upper()               # Upper- and lowercase conversions
'CODE'
>>> S.isalpha()             # Content tests: isalpha, isdigit, etc.
True

>>> line = 'aaa,bbb,ccccc,dd\n'
>>> line.rstrip()          # Remove whitespace characters on the right side
'aaa,bbb,ccccc,dd'
>>> line.rstrip().split(',') # Combine two operations, run left to right
['aaa', 'bbb', 'ccccc', 'dd']

```

Notice the last command here—it strips before it splits because Python runs it *from left to right*, making a temporary result along the way. You can chain method calls this way, as long as the prior call returns an object with methods.

Strings methods are also one way to run an advanced substitution operation known as *formatting*, available as an expression (the original), a string method call (newer), and a literal form called f-strings (newest). The first two replace keys with separate values, the third replaces embedded Python expressions with their results, and all three resolve substitution values and build new strings when they are run:

```

>>> tool = 'Python'
>>> major = 3
>>> minor = 3

>>> 'Using %s version %s.%s' % (tool, major, minor + 9)  # Format expression
'Using Python version 3.12'

>>> 'Using {} version {}.{}'.format(tool, major, minor + 9) # Format method
'Using Python version 3.12'

>>> f'Using {tool} version {major}.{minor + 9}'           # Format literal
'Using Python version 3.12'

```

And yes, this comes in *three* flavors today. While you may want to pick one for your own code, all three are fair game in code you may read (and inclusive books). Each form is rich with features, which we'll

postpone discussing until later in this book, and which tend to matter most when you must generate readable output and numeric reports:

```
>>> '%.2f | %+05d' % (3.14159, -62)           # Digits, signs, padding
'3.14 | -0062'

>>> '{1:,.2f} | {0}'.format('sapp'[1:], 296999.256)  # Commas, decimal digits
'296,999.26 | app'

>>> f'{296999.256:,.2f} | {'sapp'[1:]}'           # Ditto, with nested quotes
'296,999.26 | app'
```

Although sequence operations are generic, methods are not—while some objects share some method names, string method operations generally work only on strings, and nothing else. As a rule of thumb, Python's *toolset* is layered: generic operations that span multiple object types show up as built-in functions or expressions (e.g., `len(X)`, `X[0]`), but type-specific operations are method calls (e.g., `aString.upper()`). Finding the tools you need among all these categories will become more natural as you use Python, but the next section gives a few tips you can use right now.

Getting Help

The methods introduced in the prior section are a representative, but small, sample of what is available for string objects. In general, this book is not exhaustive in its coverage of object methods, but for more details, you can always call the built-in `dir` function. This function lists variables assigned in the caller's scope when called with no argument; more usefully, it returns a list of all the attributes available for any object passed to it. Because methods are callable attributes, they will show up in this list. Assuming `S` is still the string `'Code'`, here are its attributes:

```
>>> dir(S)
['_add_', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattr__', '__getitem__',
...etc...
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Run this live for the full list; its middle was cut at *...etc...* for space here (strings have 81 attributes today!). The names without underscores in the second half of this list are all the callable methods on string objects. The names with *double underscores* won't be important until later in the book, when we study operator overloading in classes. In short, they represent the implementation of the string object and support customization. The `__add__` method of strings, for example, is how concatenation ultimately works; Python maps the first of the following to the second internally, though you shouldn't usually use the second form yourself (it's less intuitive, and might even run slower):

```
>>> S + 'head!'
'Codehead!'
>>> S.__add__('head!')
'Codehead!'
```

The `dir` function simply gives methods' names. To ask what they do, you can pass them to the `help` function:

```
>>> help(S.replace)           # Or help(str.replace)
Help on built-in function replace:
replace(old, new, count=-1, /) method of builtins.str instance
```


Return a copy with all occurrences of substring old replaced by new.
...etc...

Press the Q key to exit a help display in most console windows. help is one of a handful of interfaces to a system of code that ships with Python known as *Pydoc*—a tool for extracting documentation from objects. Later in the book, you’ll see that Pydoc can also render its reports in HTML format for display in a web browser.

You can also ask for help on an *entire string* with help(*S*), but it may not help: the string’s value is used instead of the string itself, unless it’s empty. To do better, you need to know either the name of the string type, or that the type built-in returns any object’s type as another object: help(str) and help(type(*S*)) both give help for strings but may be more than you want—because they describe every method, help may be best used on specific methods.

For more info, you can also consult Python’s standard-library reference manual, but dir and help are the first level of documentation in Python, especially when working in an interactive REPL. Try them soon on an object near you.

Other Ways to Code Strings

So far, we’ve looked at the string object’s sequence operations and type-specific methods. Python also provides a variety of ways for us to code strings, which we’ll explore in greater depth later. For instance, special characters can be represented as *backslash escape* sequences, which Python displays in \xNN hexadecimal escape notation, unless they stand for printable characters:

```
>>> S = 'A\nB\tC'          # Escapes: \n is newline, \t is tab
>>> len(S)                  # Each stands for just one character
5

>>> S = 'A\0B\0C'          # \0, a binary zero byte, does not terminate string
>>> len(S)
5
>>> S                        # Nonprintables are displayed as \xNN hex escapes
'A\x00B\x00C'
```

As hinted earlier, Python allows strings to be enclosed in *single* or *double* quote characters—they mean the same thing but allow the other type of quote to be embedded without an escape (most programmers prefer single quotes for less clutter, unless they’re pining for other-language pasts). You can also code multiline string literals enclosed in triple quotes (single or double)—when used, all the lines are concatenated together, and newline characters (\n) are added where line breaks appear. This is useful for embedding things like multiline HTML, YAML, or JSON code in a Python script, as well as stubbing out lines of code temporarily—just add three quotes above and below:

```
>>> msg = """
... aaaaaaaaaaaaaa
... bbb'''bbb'''bbb
... cccccccc
... """
>>> msg
'\naaaaaaaaaaaaaa\nbbb'\''bbb'''bbb\ncccccccc\n'
```

Python also supports a *raw* string literal that turns off the backslash escape mechanism. Such literals start with the letter r and are useful for strings like regular-expression patterns, and directory paths on Windows sans doubled-up backslashes (e.g., r'C:\Users\you\code').

Unicode Strings

Python's strings also come with full Unicode support required for processing *non-ASCII text*. Such text includes characters in non-English languages, as well as symbols and emojis, and is common today in web pages, emails, GUIs, documents, and data. Python's string objects let you process such text seamlessly: its normal `str` string handles Unicode text (including ASCII, which is just a simple kind of Unicode); and its bytes string, together with its `bytearray` mutable cousin used earlier, handles raw byte values (including media and encoded text):

```
>>> 'hÄck'                # Normal str strings are Unicode text
'hÄck'
>>> b'a\x01c'             # bytes strings are byte-based data
b'a\x01c'
```

Formally, Python's byte strings are sequences of 8-bit bytes that print with ASCII characters when possible, and its text strings are sequences of *Unicode code points*—identifying numbers for characters, which print as the usual glyphs we've come to know and do not necessarily map to single bytes when stored in memory or encoded in files. In fact, the notion of bytes doesn't quite apply to Unicode: it includes a host of character code points too large to fit in a byte, and defines encodings for storage and transmission in which characters may be any size at all:

```
>>> 'Code'                 # Characters may be any size in memory
'Code'
>>> 'Code'.encode('utf-8') # Encoded to 4 bytes in UTF-8 in files
b'Code'
>>> 'Code'.encode('utf-16') # But encoded to 10 bytes in UTF-16
b'\xff\xfeC\x00o\x00d\x00e\x00'
```

This is especially true for richer text (`ord` gives a character's code point, and `hex` gives its hexadecimal string):

```
>>> hex(ord('🐍'))          # Code points too big for a byte
'0x1f40d'
>>> len('🐍hÄck🐍')         # One character (code point) each
6
>>> len('🐍hÄck🐍'.encode('utf-8')) # But encoded bytes sizes vary
13
>>> len('🐍hÄck🐍'.encode('utf-16'))
18
```

To code non-ASCII characters in text strings, use `\x` hexadecimal escapes; short `\u` or long `\U` Unicode escapes; or raw text interpreted per source encodings optionally declared in program files when code is read (the default for code is the universal UTF-8). To illustrate, here's our non-ASCII A-with-diaeresis character Ä coded four ways in Python:

```
>>> 'h\xc4\u00c4\u000000c4Äck' # Coding non-ASCII: hex, short, long, raw
'hÄÄÄÄck'
```

In text strings, all these forms specify Unicode *code points* that stand for characters. By contrast, byte strings use only `\x` hexadecimal escapes to embed the values of *raw bytes*; this isn't always text, but when it is, it's the encoded form of text, not its decoded code points, and encoded bytes are the same as code points only for simple text and encodings:

```
>>> '\u00A3', '\u00A3'.encode('latin1'), b'\xA3'.decode('latin1')
('£', b'\xa3', '£')
```

Apart from these string types, Unicode processing often reduces to transferring text data to and from *files*—which automatically *encode* text to bytes when stored in a file and *decode* it to characters (a.k.a.

code points) when read back into memory. Once loaded, we usually process text as strings in decoded form only. To make this work, *text files* implement encodings and accept and return text strings, but *binary files* instead deal in bytes strings for raw data.

You'll meet Unicode again in the files coverage later in this chapter, but we will save the rest of the Unicode story for later in this book. It crops up briefly in Chapters 7, 9, and 15, but for the most part is postponed until this book's advanced topics part, in Chapter 37. Unicode is crucial in many (or most) domains today, but many Python newcomers can get by with just a passing acquaintance until they've mastered string basics.

In addition to its built-in string objects, Python's standard toolset includes support for text pattern matching with its `re` module, as well as parsing textual data like JSON, CSV, XML, and HTML. You'll meet additional examples of some of these tools later in this book, but this tutorial intro has already said enough about strings and must move on.

Lists

The Python list object is the most general *sequence* provided by the language. Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size. They are also *mutable*—unlike strings, lists can be modified in place by assignment to offsets as well as a variety of list method calls. Accordingly, they provide a very flexible tool for representing arbitrary collections—files in a folder, employees in a company, emails in your inbox, and so on.

Sequence Operations

Because they are sequences, lists support all the sequence operations we discussed for strings; the only difference is that most of them return lists instead of strings. For instance, given a three-item list:

```
>>> L = [123, 'text', 1.23]           # A list of three different-type objects
>>> len(L)                             # Number of items in the list
3
```

we can index, slice, and so on, just as for strings:

```
>>> L[0]                               # Indexing by position (offset)
123
>>> L[: -1]                            # Slicing a list returns a new list
[123, 'text']

>>> L + [4, 5, 6]                       # Concat/repeat make new lists too
[123, 'text', 1.23, 4, 5, 6]
>>> L * 2
[123, 'text', 1.23, 123, 'text', 1.23]

>>> L                                   # We're not changing the original list
[123, 'text', 1.23]
```

Type-Specific Operations

Python's lists may be reminiscent of *arrays* in other languages, but they tend to be more powerful. For one thing, they have no fixed *type* constraint—the list we just looked at, for example, contains three objects of completely different types (an integer, a string, and a floating-point number). Further, lists have no fixed *size*. That is, they can grow and shrink on demand, in response to list-specific operations:

```

>>> L.append('Py')                # Growing: add object at end of list
>>> L
[123, 'text', 1.23, 'Py']

>>> L.pop(2)                      # Shrinking: delete an item in the middle
1.23
>>> L                             # "del L[2]" deletes from a list too
[123, 'text', 'Py']

```

Here, the list `append` method expands the list's size and inserts an item at the end; the `pop` method (or an equivalent `del` statement) then removes an item at a given offset, causing the list to shrink. Other list methods insert an item at an arbitrary position (`insert`), remove a given item by value (`remove`), add multiple items at the end (`extend`), and more. Because lists are mutable, most list methods also change the list object *in place*, instead of making a new one:

```

>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> M
['aa', 'bb', 'cc']
>>> M.reverse()
>>> M
['cc', 'bb', 'aa']

```

The list `sort` method here, for example, orders the list in ascending fashion by default, and `reverse` reverses it; in both cases, the methods modify the list directly (though similarly named functions we'll explore later do not).

Bounds Checking

Although lists have no fixed size, Python still doesn't allow us to reference items that are not present. Indexing off the end of a list is always a mistake, but so is assigning off the end (error messages are condensed here and elsewhere):

```

>>> L
[123, 'text', 'Py']

>>> L[99]
IndexError: list index out of range

>>> L[99] = 1
IndexError: list assignment index out of range

```

This is intentional, as it's usually an error to try to assign off the end of a list (and a particularly nasty one in the C language, which doesn't do as much error checking as Python). Rather than silently growing the list in response, Python reports an error. To grow a list, we call list methods such as `append` instead (or make a new list). Unlike indexing, slicing *scales* offsets to be in bounds, but this will have to await coverage in the in-depth chapters ahead.

Nesting

One nice feature of Python's core object types is that they support arbitrary *nesting*—we can nest them in any combination, and as deeply as we like. For example, we can have a list that contains a dictionary, which contains another list, and so on—as deeply and mixed as needed to describe things in our real world.

An immediate application of this feature is to represent matrixes, or “multidimensional arrays” in Python. A list with nested lists like the following will do the job for basic applications (coding fine print: indentation doesn't matter in this, Python expressions with unclosed brackets can span multiple lines this way, and some REPLs show “...” continuation-line prompts, which are often omitted in this book for easier copy and paste from emedia):

```
>>> M = [[1, 2, 3],          # A 3 × 3 matrix, as nested lists
          [4, 5, 6],        # Code can span lines if bracketed
          [7, 8, 9]]

>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Here, we've coded a list that contains three other lists. The effect is to represent a 3×3 matrix of numbers. Such a structure can be accessed in a variety of ways:

```
>>> M[1]                # Get row 2
[4, 5, 6]

>>> M[1][2]             # Get row 2, then get item 3 within the row
6
```

The first operation here fetches the entire second row, and the second grabs the third item within that row (it runs left to right, like the earlier string strip and split combo). Stringing together index operations takes us deeper and deeper into our nested-object structure. Tip: this matrix structure works for small-scale tasks, but for more serious number crunching you will probably want to use the open source *NumPy* extension for Python, which can store and process large matrixes much more efficiently than our nested list structure (and is well out of scope here; see [Chapter 1](#)).

Comprehensions

In addition to sequence operations and list methods, Python includes a more advanced operation known as a *list comprehension* expression, which turns out to be a powerful way to process structures like our matrix. Suppose, for instance, that we need to extract the second column of the prior section's matrix. It's easy to grab rows by simple indexing because the matrix is stored by rows, but it's almost as easy to get a *column* with a list comprehension:

```
>>> col2 = [row[1] for row in M]          # Collect the items in column 2
>>> col2
[2, 5, 8]

>>> M                                     # The matrix is unchanged
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

List comprehensions are a way to build a new list by running an expression on each item in a sequence, one at a time, from left to right. They are coded in square brackets (to tip you off to the fact that they make a list) and are composed of an expression and a looping construct that share a variable name (row, here). The preceding list comprehension means basically what it says: “Give me row[1] for each row in matrix M, in a new list.” The result is a new list containing column 2 of the matrix.

List comprehensions can be more complex in practice:

```
>>> [row[1] + 1 for row in M]             # Add 1 to each item in column 2
[3, 6, 9]

>>> [row[1] for row in M if row[1] % 2 == 0] # Filter out odd items (pick evens)
[2, 8]
```

The first operation here, for instance, adds 1 to each item as it is collected, and the second uses an *if* clause to *filter* odd numbers out of the result using the % modulus expression (remainder of division). List comprehensions make new lists of results, but they can be used to iterate over any *iterable* object—a term we'll flesh out later in this book, but which simply means either a physical sequence or a virtual one that produces its items on request. Here, for instance, we use list comprehensions to step over a hardcoded list of coordinates and a string:

```
>>> diag = [M[i][i] for i in [0, 1, 2]]      # Collect a diagonal from matrix
>>> diag
[1, 5, 9]

>>> doubles = [c * 2 for c in 'hack']        # Repeat characters in a string
>>> doubles
['hh', 'aa', 'cc', 'kk']
```

These expressions can also be used to collect multiple values, as long as we wrap those values in a nested collection. The following illustrates using *range*—a built-in that generates successive integers and requires a surrounding *list* to force it to yield all its values for display in the REPL (there's more on this mystery ahead):

```
>>> list(range(4))                          # Integers 0..(N-1)
[0, 1, 2, 3]
>>> list(range(-6, 7, 2))                   # -6 to +6 by 2
[-6, -4, -2, 0, 2, 4, 6]

>>> [[x ** 2, x ** 3] for x in range(4)]    # Multiple values, "if" filters
[[0, 0], [1, 1], [4, 8], [9, 27]]

>>> [[x, x // 2, x * 2] for x in range(-6, 7, 2) if x > 0]
[[2, 1, 4], [4, 2, 8], [6, 3, 12]]
```

As you can probably tell, list comprehensions are too involved to cover more formally in this preview. The main point of this brief introduction is to illustrate that Python includes both simple and advanced tools in its arsenal. List comprehensions are optional, but they can be very useful in practice and often provide a processing speed advantage.

In fact, comprehension syntax is not just for making lists: enclosing it in *parentheses* can also be used to create an iterable object known as a *generator*, which produces results on demand per Python's *iteration protocol*—in the following, summing items in a matrix row with the built-in *sum*, on each call to *next*:

```
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> G = (sum(row) for row in M)              # Make a generator of row sums
>>> next(G)                                 # Run the iteration protocol (ahead)
6
>>> next(G)                                 # A new row sum on each call
15
>>> next(G)                                 # Row 3: 7 + 8 + 9
24
```

And comprehension syntax can also be used to create *sets* and *dictionaries* when enclosed in curly braces:

```
>>> {sum(row) for row in M}                # Makes an unordered set of row sums
{24, 6, 15}

>>> {i: sum(M[i]) for i in range(3)}        # Makes key:value table of row sums
{0: 6, 1: 15, 2: 24}
```

But to grasp concepts like the iteration protocol and objects like sets and dictionaries, we must move ahead.

Dictionaries

Unlike strings and lists, Python dictionaries are not sequences at all but are instead the only core member of a category known as *mappings*. Mappings are also collections of other objects, but they store objects by *key* instead of by relative position. While dictionaries retain the insertion order of their keys today, this may not apply to your goals, and key-to-value mapping remains their main role. Dictionaries are also *mutable*: like lists, they may be changed in place and can grow and shrink on demand. Also like lists, they are a flexible tool for coding collections, but their more *mnemonic* keys are better suited when a collection's items are named or labeled—as in fields of a database record.

Mapping Operations

When written as literals, dictionaries are coded in curly braces and consist of a series of “*key: value*” pairs. Dictionaries are useful anytime we need to associate a set of values with keys—to describe the properties of something. As an example, consider the following three-item dictionary with keys “name,” “job,” and “age,” that record the attributes of a fictitious (and wholly generic and neutral) worker:

```
>>> D = {'name': 'Pat', 'job': 'dev', 'age': 40}
```

We can index this dictionary by key to fetch and change its keys’ associated values. The dictionary index operation uses the same syntax as that used for sequences, but the item in the square brackets is a key, not a relative position:

```
>>> D['name']                # Fetch value of key 'name'
'Pat'

>>> D['job'] = 'mgr'          # Change Pat's job description
>>> D
{'name': 'Pat', 'job': 'mgr', 'age': 40}
```

Although the curly-braces literal form does see use, it is perhaps more common to see dictionaries built up in different ways (after all, it’s rare to know all your program’s data before your program runs). The following code, for example, starts with an empty dictionary and fills it out one key at a time. Unlike out-of-bounds assignments in lists, which are forbidden, assignments to new dictionary keys create those keys:

```
>>> D = {}
>>> D['name'] = 'Pat'          # Create keys by assignment
>>> D['job'] = 'dev'
>>> D['age'] = 40
>>> D
{'name': 'Pat', 'job': 'dev', 'age': 40}
```

Here, we’re effectively using dictionary keys as field names in a record that describes an imaginary person. In other roles, dictionaries can also be used to replace *searching* operations—indexing a dictionary by key is often the fastest way to code a search in Python.

As you’ll learn later, we can also make dictionaries by passing to the `dict` type name either *keyword arguments* (a special *name=value* syntax in function calls), or the result of *zipping* together sequences of keys and values obtained at runtime (e.g., from files). Both of the following make the same dictionary as the prior example and its equivalent `{}` literal form; the first requires string keys but tends to make for less typing (and subjective noise), and the second uses the `zip` built-in we’ll study later in this part of the book:

```
>>> pat1 = dict(name='Pat', job='dev', age=40)           # Keywords
>>> pat1
{'name': 'Pat', 'job': 'dev', 'age': 40}

>>> pat2 = dict(zip(['name', 'job', 'age'], ['Pat', 'dev', 40])) # Zipping
>>> pat2
{'name': 'Pat', 'job': 'dev', 'age': 40}
```

Notice how the left-to-right order of dictionary keys is always the same. Though not sequences, dictionary keys retain their *insertion order*, even in the presence of changes: the order of keys is the order in which keys were added. This wasn’t the norm until Python 3.7, and prior to this, key order was scrambled and sometimes required separate ordering. The new ordering may be more intuitive and will be assumed in this book but adds a sequence flavor to dictionaries not shared by other nonsequences. Like most mods, it also rewrites history and invalidates Python learning resources that cannot be updated as frequently as Python. Change is almost always a double-edged sword.

Nesting Revisited

In the prior example, we used a dictionary to describe a hypothetical person, with three keys. Suppose, though, that the information is more complex. Perhaps we need to record a first name and a last name, along with multiple job titles. This leads to another application of Python’s object nesting in action. The following dictionary, coded all at once as a literal, captures more-structured information (again, indentation here and “...” in some REPLs are moot):

```
>>> rec = {'name': {'first': 'Pat', 'last': 'Smith'},
          'jobs': ['dev', 'mgr'],
          'age': 40.5}
```

Here, we again have a three-key dictionary at the top (keys “name,” “jobs,” and “age”), but the values have become more complex: a nested dictionary for the name to support multiple parts, and a nested list for the jobs to support multiple roles and future expansion. We can access the components of this structure much as we did for our list-based matrix earlier, but this time most indexes are dictionary keys, not list offsets:

```
>>> rec['name']           # 'name' is a nested dictionary
{'first': 'Pat', 'last': 'Smith'}

>>> rec['name']['last']   # Index the nested dictionary
'Smith'

>>> rec['jobs']           # 'jobs' is a nested list
['dev', 'mgr']
>>> rec['jobs'][-1]       # Index the nested list
'mgr'

>>> rec['jobs'].append('janitor') # Expand Pat's job description in place
>>> rec
{'name': {'first': 'Pat', 'last': 'Smith'}, 'jobs': ['dev', 'mgr', 'janitor'], 'age': 40.5}
```


Notice how the last operation here *expands* the nested jobs list—because the jobs list is a separate piece of memory from the dictionary that contains it, it can grow and shrink freely (the `pop` method we met earlier is one way to shrink objects). This may seem quite a trick to readers with backgrounds in more rigid languages, but is natural in Python.

More fundamentally, this example demos the *flexibility* of Python’s core object types. As you can see, nesting allows us to build up complex information structures directly and easily. Building a similar structure in a low-level language like C would be tedious and require much more code: we would have to lay out and declare structures and arrays, fill out values, link everything together, and so on. In Python, this is all automatic—running the expression creates the entire nested object structure for us. In fact, this is one of the main benefits of scripting languages like Python.

Just as importantly, in lower-level languages we may have to allocate *memory* space for objects ahead of time and be careful to release it when we no longer need it. In Python, this is all automatic: object memory is allotted as needed and freed when we lose the last reference to the object—by assigning its variable to something else, for example:

```
>>> rec = 0          # Now the prior object's space is reclaimed
```

Technically speaking, Python uses a scheme called *garbage collection* that reclaims unused memory as your program runs and frees you from having to manage such details in your code. In standard Python (a.k.a. CPython) this uses object *reference counts* primarily, along with a supplemental garbage collector for cycles. We’ll study how this works later in [Chapter 6](#); for now, it’s enough to know that you can use objects freely, while Python handles their memory.

Watch for a record structure similar to the one we just coded in Chapters 8, 9, and 27, where we’ll use it to compare and contrast lists, dictionaries, tuples, named tuples, and classes—an array of data structure options with trade-offs we’ll cover in full later. It’s also worth noting that the record structure we used here can be saved in a file with a variety of techniques in Python, including its `pickle` module and support for language-neutral *JSON* (a data format that’s strikingly similar to Python dictionary objects); more on such tools later in this book.

Missing Keys: if Tests

As mappings, dictionaries support accessing items by key only, with the sorts of operations we’ve just seen. In addition, though, they also support type-specific operations with *methods* that are useful in a variety of common roles. For example, the dictionary `copy` and `update` methods copy a dictionary and merge one into another in place, respectively (the `|` operator does the same, but makes a new dictionary for its result: it’s just a copy plus an update).

Dictionary methods also play parts in common key use cases. For instance, while we can assign to a new key to expand a dictionary, fetching a nonexistent key is still a mistake:

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> d['d'] = 4          # Assigning new keys grows dictionaries
>>> d
{'a': 1, 'b': 2, 'c': 3, 'd': 4}

>>> d['e']             # Referencing a nonexistent key is an error
...error text omitted...
KeyError: 'e'
```

This is what we want—it’s usually a programming error to fetch something that isn’t really there. But in generalized programs, we can’t always know what keys will be present when we write our code. How do we handle such cases and avoid errors? One solution is to test ahead of time. The dictionary in

membership expression allows us to query the existence of a key and branch on the result with a Python `if` statement.

Which brings us to our first Python *compound statement*—one with nested parts. If you’re working along, here are a few practical bits: in the following, press the Enter key twice to run the `if` interactively after typing its code (an empty line means “go” in most REPLs, as explained in [Chapter 3](#)); the prompt changes to a “...” after the first line in some interfaces (as for the earlier multiline dictionaries and lists); and indentation matters this time (for reasons up next):

```
>>> 'e' in D                                # Boolean result: True or False (see ahead)
False

>>> if not 'e' in D:                        # Python's main selection statement
    print('missing key!')

missing key!
```

This book has more to say about the `if` statement in later chapters, but its syntax is straightforward. It consists of the word `if`, followed by an expression whose result is interpreted as true or false, followed by a block of code to run if the expression result is true. In its full regalia, the `if` statement can also have an `else` clause for the false case, and one or more `elif` (“else if”) clauses for other tests.

Functionally, the `if` is the main *selection* tool in Python, and how we code most of the logic of choices and decisions in our scripts. It’s joined by its ternary `if/else` expression cousin you’ll meet in a moment, the `if` comprehension filter lookalike we used earlier, and the newer `match` multiple-selection statement you’ll meet later in this book.

If you’ve used some other programming languages in the past, you might now be wondering how Python knows when the `if` statement ends. We’ll explore Python’s syntax rules in depth in later chapters, but in short, if you have more than one action to run in a statement block, you simply indent all their statements the same way—which both promotes readable code and reduces the number of characters you have to type. All multiline statements follow this pattern: a header line ending in “:” and a block of (usually) indented code, with no “{” around blocks, and no “;” required after statements (though fair warning: forgetting the “:” is the most common beginner’s mistake in Python!):

```
>>> if not 'e' in D:
    print('missing')           # Statement blocks are indented
    print('no, really...')     # (Unless they're simple: see ahead)

missing
no, really...
```

Besides the `in` test, there are a variety of other ways to avoid accessing nonexistent keys in the dictionaries we create: the dictionary `get` method, which is a conditional index with a default; the `if/else` ternary (three-part) expression, which is essentially a limited `if` statement squeezed onto a single line; and the `try` statement, which is a tool we’ll first use in [Chapter 10](#) that catches and recovers from errors altogether. Here are the first two in action:

```
>>> D.get('a', 'missing')          # Like D['a'] but with a default
1
>>> D.get('e', 'missing')          # Default returned if absent
'missing'

>>> D['e'] if 'e' in D else 0       # if/else ternary expression form
0
```

We'll save the details on such alternatives until a later chapter. For now, let's turn to other dictionary methods' roles in another common use case.

Item Iteration: for Loops

Dictionaries collect a lot of useful info, but what do we do if we need to process their items one at a time? As it turns out, dictionaries come with methods designed for the job:

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'b': 2, 'c': 3}

>>> list(D.keys())           # Keys, values, and key/value pairs
['a', 'b', 'c']
>>> list(D.values())         # list forces results generation
[1, 2, 3]
>>> list(D.items())
[('a', 1), ('b', 2), ('c', 3)]
```

As shown, a dictionary's `keys`, `values`, and `items` methods return its keys, values, and key/value pairs (the latter is tuples, up next on this tour). Really, though, these methods all return an object that produces results one at a time, which is why they've been wrapped in `list` calls, as we did for `range` earlier. This reflects the *iteration protocol* in Python—a concept we'll explore in full later, but which boils down to an iterable object, which has an iterator object, which responds to next calls to produce one result at a time:

```
>>> D.keys()                 # Get an iterable object
dict_keys(['a', 'b', 'c'])

>>> I = iter(D.keys())       # Get an iterator from an iterable
>>> next(I)                   # Get one result at a time from iterator
'a'
>>> next(I)                   # This is most of the iteration protocol
'b'
```

We used the same `next` built-in to force results from a generator comprehension earlier, but without the `iter` step: because generators don't support multiple scans, they are their own iterators, and `iter` is a no-op.

Tools that support this protocol can both save memory and minimize delays, because they don't produce all their results at once. The iteration protocol works on all sorts of objects in Python, but you can usually forget its details if you use the Python `for` loop, which runs the iteration protocol automatically to step through items one at a time—both for physical collections like strings and lists, and virtual sequences like generators, `range`, and `keys`:

```
>>> for key in D.keys():     # Auto run the iteration protocol
    print(key, '=>', D[key]) # Display multiple items, space between

a => 1
b => 2
c => 3
```

To code a `for`, provide a variable (e.g., `key`) and an iterable object (e.g., `D.keys()`); for each item in the object, the `for` assigns the item to the variable and runs the nested (and usually indented) block of code—which uses the variable to refer to the current item each time through. The `for` is one of the

main *repetition* tools in Python, together with the comprehensions you met earlier, and the more general-purpose while loop you'll meet later in this book.

Because dictionary iteration is so common, the `for`, and similar iteration tools, can also step through keys implicitly, as well as key/value pairs. The following loops, for example, produce the same output as the preceding example; the choice between all these forms is partly a matter of personal preference, though explicit is generally better:

```
>>> for key in D:                # Implicit keys() iteration
    print(key, '=>', D[key])

>>> for (key, value) in D.items(): # Key/value-pair tuples iteration
    print(key, '=>', value)
```

The last of these uses something known as *tuple assignment*, which automatically unpacks items into variables. But to fully understand the sorts of stuff we get back from the dictionary `items` method, we have to move ahead.

Tuples

The tuple object (pronounced “toople” or “tuhple,” depending on whom you ask) is roughly like a list that cannot be changed—tuples are *sequences*, like lists, but they are *immutable*, like strings. Functionally, they're used to represent fixed collections of items: the components of a specific calendar date, for instance. Syntactically, they are normally coded in parentheses instead of square brackets and support arbitrary object types, arbitrary nesting, and the usual sequence operations that we used on strings and lists earlier:

```
>>> T = (1, 2, 3, 4)            # A 4-item tuple
>>> len(T)                      # Length
4

>>> T + (5, 6)                  # Concatenation: a new tuple
(1, 2, 3, 4, 5, 6)

>>> T[0], T[1:]                 # Indexing, slicing, and more
(1, (2, 3, 4))
```

As usual, tuples also have type-specific callable methods, but not nearly as many as lists:

```
>>> T.index(4)                  # Tuple methods: 4 appears at offset 3
3
>>> T.count(4)                  # 4 appears once
1
```

The primary distinction for tuples is that they cannot be changed once created. That is, they are immutable sequences (quirk: one-item tuples like the one here require a trailing comma to distinguish them from simple expressions):

```
>>> T[0] = 2                    # Tuples are immutable
TypeError: 'tuple' object does not support item assignment

>>> T = (2,) + T[1:]            # Make a new tuple for a new value
>>> T
(2, 2, 3, 4)
```

Like lists and dictionaries, tuples support mixed types and nesting, but they don't grow and shrink like lists and dictionaries because they are immutable:

```
>>> T = 'hack', 3.0, [11, 22, 33]
>>> T
('hack', 3.0, [11, 22, 33])

>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
AttributeError: 'tuple' object has no attribute 'append'
```

Notice the first line in this: the *parentheses* enclosing a tuple's items can often be omitted, as done here. In contexts where commas don't otherwise matter, the *commas* are what actually builds a tuple. This also explains why REPLs show results in parentheses when you enter multiple items separated by commas: the input is really a tuple.

Why Tuples?

So, why have a kind of object that is like a list, but supports fewer operations? Frankly, tuples are not used as often as lists in practice, but their immutability is the whole point. If you pass a collection of objects around your program as a list, it can be changed anywhere; if you use a tuple, it cannot. That is, tuples provide a sort of integrity constraint that is convenient in programs larger than those here. We'll talk more about tuples later in the book, including an extension that builds upon them called *named tuples*. For now, though, let's move on to this tour's last major object.

Files

File objects are the main way your Python code will access the content of files on your computer. They can be used to read and write text memos, audio clips, Excel documents, saved emails, and whatever else you have stored on your device. Files are a core object type, but they're something of an oddball—there is no specific literal syntax for creating them. Rather, you create a file object by calling the built-in `open` function with an external filename, and perhaps more depending on your goals.

For example, to create a text output file, pass in its name and the 'w' processing mode string to *write* text data; Python automatically makes the newline character `\n` portable across platforms when it's transferred to and from files:

```
>>> f = open('data.txt', 'w')      # Open a new file in text-output mode
>>> f.write('Hello\n')             # Write strings of characters to it
6
>>> f.write('world!\n')           # Return number of items written
7
>>> f.close()                     # Close to flush output buffers to disk
```

This creates a file in the current directory and writes text to it (the filename can be a full directory path if you need to access a file elsewhere on your device). To read back what you just wrote, reopen the file in 'r' processing mode, for *reading* text input—this is also the default if you omit the mode in the call. Then read the file's content into a string and access it. A file's content is always a string in your script, regardless of the type of data the file contains:

```

>>> f = open('data.txt')           # Open an existing file in text-input mode
>>> text = f.read()                 # Read entire file into a string
>>> text
'Hello\nworld!\n'

>>> print(text)                     # print interprets control characters
Hello
world!

>>> text.split()                    # File content is always a string
['Hello', 'world!']

```

Other file object methods support additional features we don't have time to cover here. For instance, file objects provide more ways of reading and writing (`read` accepts an optional maximum byte/character size, `readline` reads one line at a time, and so on), as well as other tools (`seek` moves to a new file position). As you'll see later, though, the best way to read a text file is usually to not read it at all—files support the *iteration protocol* with an iterator that automatically reads line by line in `for` loops and other contexts:

```

>>> for line in open('data.txt'):    # Display lines in a file
    print(line.rstrip())             # Single spaced (sans \n)

```

You'll meet the full set of file methods later in this book, but if you want a quick preview now, run a `dir` call on any open file and a `help` on any of the method names that come back, as we learned earlier.

Unicode and Byte Files

The prior section's examples illustrate file basics that suffice for many roles. Technically, though, they rely on the platform's Unicode *encoding* default for the host platform. Python text files always use a Unicode encoding to encode strings on writes and decode them on reads. This is often irrelevant for simple ASCII text data, which usually maps to and from file bytes unchanged. But for richer kinds of data, file interfaces can vary by content type.

As hinted when we studied Unicode strings earlier, Python draws a sharp distinction between text and binary data in files: *text files* represent content as normal `str` strings and perform Unicode encoding and decoding automatically as noted, while *binary files* represent content as the special bytes string and allow you to access file content unaltered.

For example, *binary files* are useful for processing media, accessing data created by C programs, and so on. What you send and receive is the literal content of the file, whether it's encoded text or a JPEG image. Add a `b` to the mode to invoke binary (and expect a `\r\n` instead of `\n` at the end on Windows, because its newlines vary from Unix):

```

>>> bf = open('data.bin', 'wb')
>>> bf.write(b'h\xFFa\xEEc\xDDk\n')  # Write binary data in a bytes
8
>>> bf.close()
>>> open('data.bin', 'rb').read()     # Read binary data to a bytes
b'h\xffa\xeec\xddk\n'

```

For *text files*, we can't really talk about content without also asking, “What kind?”—files may use any Unicode encoding, especially if they came from another platform, or the internet at large. This applies to portable programs too: if you want your code to work across platforms, you should generally make encodings explicit to avoid unpleasant surprises. Luckily, this is easier than it may sound—simply pass in an encoding name to open to force an encoding:

```
>>> tf = open('unidata.txt', 'w', encoding='utf-8')
>>> tf.write('🐍h\u00c4ck🐍')          # Encodes to UTF-8
6
>>> tf.close()
```

If you read with the same encoding (or one that's compatible), you get back the same text-character code points that you wrote. The encoded bytes on the file are in UTF-8 form, but your code usually doesn't need to care:

```
>>> open('unidata.txt', 'r', encoding='utf-8').read()    # Decodes from UTF-8
'🐍hAck🐍'

>>> open('unidata.txt', 'rb').read()                    # Raw encoded text
b'\xf0\x9f\x90\x8d\xc3\x84ck\xf0\x9f\x91\x8f'
```

While files automate most encodings, you can also encode and decode manually if your program gets Unicode data from another source—parsed from an email message or fetched over a network connection, for example:

```
>>> 'hÄck'.encode('utf-8')
b'h\xc3\x84ck'
>>> b'h\xc3\x84ck'.decode('utf-8')
'hÄck'
```

Python also supports non-ASCII file *names* (not just content), but it's largely automatic. For the whole story on Unicode in Python, stay tuned for [Chapter 37](#).

Other File-Like Tools

The `open` function is the workhorse for most file processing you will do in Python. For more advanced tasks, though, Python comes with additional file-like tools: pipes, FIFOs, sockets, keyed-access files, persistent object shelves, descriptor-based files, relational and object-oriented database interfaces, and more. We won't cover many of these topics in this language book, but you'll find them useful once you start programming Python in earnest.

Other Object Types

Beyond the core object types we've seen so far, there are others that get less publicity than their cohorts but are useful in their intended roles nonetheless. Let's quickly run down some of the stragglers in this category.

Sets

Python sets are neither mappings nor sequences; rather, they are unordered collections of immutable (technically, “hashable”) objects, which store each object just once. You create sets by calling the built-in `set` function with a sequence or other iterable, or by using a set literal expression, and sets support the usual mathematical set operations:

```
>>> X = set('hack') # Sequence => set
>>> Y = {'a', 'p', 'p'} # Set literal
>>> X, Y
({'c', 'k', 'a', 'h'}, {'p', 'a'})

>>> X & Y, X | Y # Intersection, union
({'a'}, {'a', 'p', 'c', 'k', 'h', 'a'})

>>> X - Y, X > Y # Difference, superset
({'c', 'k', 'h'}, False)
```

Even less mathematically inclined programmers often find sets useful for common tasks such as filtering out duplicates, isolating differences, and performing order-neutral equality tests without sorting:

```
>>> list(set([3, 1, 2, 1, 3, 1]))      # Duplicates removal
[1, 2, 3]
>>> set('code') - set('hack')          # Collection difference
{'d', 'o', 'e'}
>>> set('code') == set('deoc')         # Order-neutral equality
True
```

As you'll see later in this part of the book, normal sets themselves are mutable and can be changed (with their `remove` and `add` methods, for example), though the immutable items within them by definition cannot.

Booleans and None

Python also comes with Booleans, with predefined `True` and `False` objects that are essentially just the integers 1 and 0 with custom display logic; as well as a special placeholder object called `None`, commonly used to initialize names and objects and designate an absence of a result in functions:

```
>>> 1 > 2, 1 < 2                # Booleans
(False, True)

>>> bool('hack')                 # All objects have a Boolean value
True                             # Nonempty means True

>>> X = None                     # None placeholder
>>> print(X)                     # But None is a thing
None

>>> L = [None] * 100             # Initialize a list of 100 Nones
>>> L
[None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, ...etc.: a list of 100 Nones...]
```

Types

One last core object merits a callout here. The *type* object, returned by the `type` built-in function, is an object that gives the type of another object. We used it earlier when exploring the `help` function, but here's its actual result:

```
>>> L = [1, 2, 3]
>>> type(L)           # The type of a list object
<class 'list'>
>>> type(type(L))     # Even types are objects!
<class 'type'>
```


Besides allowing you to explore your objects interactively, the `type` object in its most practical application allows code to check the types of the objects it processes. In fact, there are at least three ways to do so in a Python script:

```
>>> type(L) == type([])          # Type testing, if you must...
True                             # Using a real object

>>> type(L) == list              # Using a type name
True

>>> isinstance(L, list)          # The object-oriented way
True
```

But now that this book has shown you all these ways to do type testing, it's required by Python law to tell you that doing so is almost always the wrong thing to do in a Python program (and often a sign of an ex-Java programmer first starting to use Python!). The reason won't become completely clear until later in the book when we start writing larger code units like functions, but it's a—and perhaps *the*—core Python concept. By checking for specific types in your code, you effectively break its flexibility: you limit it to working on just one type. Without such checks, your code may be able to work on a whole range of types automatically.

This is part of the *polymorphism* mentioned earlier, and it stems from Python's lack of type declarations. As you'll learn more when we step up to coding functions and classes, in Python, we code to object *interfaces* (operations supported), not to types. That is, we care what an object *does*, not what it *is*. Not caring about specific types means that code can be applied to many of them: any object with a compatible interface will work, regardless of its specific type. Although type checking is supported—and even required in some rare cases—you'll see that it's not usually the “Pythonic” way of thinking. In fact, you'll probably find that polymorphism is the key to using Python well.

Type Hinting

That being said, Python has slowly accumulated a type-declaration facility known as *type hinting*, based originally on its earlier function annotations and inspired by the *TypeScript* dialect of JavaScript. With these syntax and module extensions, it is possible to name expected object types of function arguments and results, attributes in class-based objects, and even simple variables in Python code, and these hints may be used by external type checkers like *mypy*:

```
>>> x: int = 1                  # Optional hint: x might be an integer
>>> x = 'anything'             # But it doesn't have to be!
```

Importantly, though, Python type hinting is meant only for documentation and use by third-party tools. The Python language *does not itself mandate or use type declarations* and has no plans to ever do so. Hence, type hinting by most measures is largely academic, no more useful than in-program comments, and oddly contrary to Python's core ideas. The fact that it was nevertheless elevated to language syntax and complex subdomain arguably does a disservice to Python learners and users alike. Especially for beginners, this is an optional and peripheral topic that's best deferred until you master the flexibility of Python's dynamic typing. We'll study it only briefly in this book, in [Chapter 6](#).

To be sure, type hinting does not mean that Python is no longer dynamically typed. Indeed, a statically typed Python that requires type declarations would not be a Python at all! Some programmers accustomed to restrictive languages may regrettably code Python type hints anyhow as a hard-to-break habit (or misguided display of prowess), but good programmers focus instead on Python's polymorphism. As you'll find, it's how to code Python in Python.

User-Defined Objects

We won't study *object-oriented programming* (OOP) in Python and its `class` statement until later in this book. In abstract terms, though, classes define new types of objects that extend the core set, so they merit a passing glance here. Suppose, for example, that you wish to have a kind of object that models workers in a company. Although there is no such specific core object type in Python (it's not an HR language, after all), a user-defined class might fit the bill:

```
>>> class Worker:
    ...stay tuned for Part VI...
```

Most of this code is omitted because it wouldn't make much sense at this point in the book, but such a class might define *attributes* of workers like `name` and `pay`, as well as *behavior* coded as custom methods. Calling the class would generate objects that are instances of our new type, and the class's methods would process them:

```
>>> sue = Worker('Sue Jones', 60000)      # Make two new objects
>>> bob = Worker('Bob Smith', 50000)      # Each has a name and pay
>>> sue.lastName()                        # Call a method to process sue
'Jones'
>>> bob.lastName()                       # Call a method to process bob
'Smith'
>>> sue.giveRaise(.10)                   # Update sue's pay
>>> sue.pay                               # Display sue's pay
66000.0
```

This is called *object-oriented*, because there is always an implied subject in functions within a class. Class-based objects ultimately use built-in objects internally, and we can always describe things like workers with Python's built-in objects instead, as we did with dictionaries and lists earlier. Classes, though, implement operations with meaningful names, add structure to your code, and come with inheritance mechanisms that lend themselves to customization by *extension*. In OOP, we strive to extend software by writing new classes, not by changing what already works.

All of which is well beyond the bounds of this object preview, though, so we must stop short here. For full disclosure on user-defined object types coded with classes, you'll have to read on. Because classes build upon other tools in Python, they are one of the major destinations of this book's journey.

And Everything Else

As mentioned earlier, everything you can process in a Python script is a type of object, so our object-type tour is necessarily incomplete. However, even though everything in Python is an “object,” not everything is considered a part of Python's *core* toolset. Other object types in Python either are related to program execution (like functions, modules, classes, and compiled code), or are implemented by imported module functions, not language syntax. The latter of these also tend to have application-specific roles—text patterns, database interfaces, network connections, and so on.

Moreover, keep in mind that the objects we've met here are *objects*, but not necessarily *object-oriented*—a concept that usually requires the Python `class` statement, which you'll meet again later in this book. Still, Python's core objects are the workhorses of all Python scripts you're likely to meet and are often the basis of larger noncore objects.

Chapter Summary

And that's a wrap for our object tour. This chapter has previewed Python's core object types and the sorts of operations we can apply to them. We've studied generic operations that work on many object types (sequence operations such as indexing and slicing, for example), as well as type-specific operations available as method calls (string splits and list appends, for instance). We've also defined some key terms, such as immutability, sequences, and polymorphism.

Along the way, we've learned that Python's core object types are more flexible and powerful than what is available in lower-level languages. For instance, Python's lists and dictionaries can nest, grow and shrink, and contain objects of any type, and their space is automatically created and cleaned up as you go. We've also glimpsed the ways that strings and files work hand in hand to support binary and text data, peeked at the iteration protocol and OOP, discussed the perils of type hinting, and introduced the `if` and `for` statements we'll be using ahead.

This chapter skipped many of the details in order to provide a first tour, so you shouldn't expect all of this chapter to have made sense yet. In the next few chapters, we'll start to dig deeper, taking a second pass over Python's object types that will fill in details omitted here and give you a deeper understanding. We'll start off the next chapter with an in-depth look at Python numbers. First, though, here is another quiz to review.

Test Your Knowledge: Quiz

We'll explore the concepts introduced in this chapter in more detail in upcoming chapters, so we'll just cover the big ideas here:

1. Name four of Python's core object types.
2. Why are they called "core" object types?
3. What does "immutable" mean, and which three of Python's object types are considered immutable?
4. What does "sequence" mean, which objects fall into this category, and how is "iterable" related?
5. What does "mapping" mean, and which core object type is a mapping?
6. What is "polymorphism," and why should you care?

Test Your Knowledge: Answers

1. Numbers, strings, lists, dictionaries, tuples, files, and sets are generally considered to be the core object types. Booleans, `None`, and types themselves are classified this way as well. Some of these types are really categories: there are multiple number types (integer, floating point, complex, fraction, and decimal) and multiple string types (text strings, byte strings, and mutable byte strings).
2. They are known as "core" object types because they are part of the Python language itself and are always available. To create other objects, you generally must call functions in imported modules. Most of the core objects have specific syntax for generating their objects: `'hack'`, for example, is an expression that makes a string and determines the set of operations that can be applied to it. Because of this, core objects are hardwired into Python's syntax. In contrast, you must call the built-in `open` function to create a file object, even though this is usually considered a core object type too.

3. An “immutable” object is an object that cannot be changed after it is created. Numbers, strings, and tuples in Python fall into this category. While you cannot change an immutable object in place, you can always make a new one by running an expression. `bytearrays` offer mutability for strings, but they only apply directly to text if it’s a simple 8-bit kind (e.g., ASCII).
4. A “sequence” is a positionally ordered collection of objects. Strings, lists, and tuples are all sequences in Python. They share common sequence operations, such as indexing, concatenation, and slicing, but also have type-specific method calls. The related term “iterable” means either a physical sequence, or a virtual one that produces its items on request. Sequences are iterable, but so are generators, files, results of functions like `range`, and the dictionary object (which produces its keys when iterated, just like its `keys` method).
5. The term “mapping” denotes an object that maps keys to associated values. Python’s dictionary is the only mapping among its core object types. Mappings retain insertion order (as of Python 3.7), and support access to data stored by key, plus type-specific method calls that enable key tests, iteration, and more.
6. “Polymorphism” means that the meaning of an operation (like a `+`) depends on the objects being operated on. This turns out to be a key idea (and perhaps the largest) behind using Python well—not constraining code to specific types makes that code automatically applicable to many types. This becomes more obvious when coding functions and classes in Python. While Python today has type hinting, it’s not used by Python itself and is meant only for documentation and tools.

Numbers and Expressions

This chapter begins our in-depth tour of the Python language. In Python, data takes the form of *objects*—either built-in objects that Python provides or objects we create using Python tools and other languages like C. In fact, objects are the basis of every Python program you will ever write. Because they are the most fundamental notion in Python programming, objects are also our first focus in this book.

In the preceding chapter, we took a quick first pass over Python's core object types. Although essential terms were introduced in that chapter, we avoided covering too many specifics in the interest of space. Here, we'll begin a more careful second look at object concepts, to fill in details we glossed over earlier. Let's get started by exploring our first category: Python's numeric objects and operations.

Numeric Object Basics

Most of Python's numeric support is fairly typical and will probably seem familiar if you've used almost any other programming language in the past. They can be used to keep track of your bank balance, the distance to Mars, the number of visitors to your website, and just about any other numeric quantity.

In Python, numbers are not really a single object type, but a category of similar types. Python supports the usual numeric types (integers and floating points), as well as literals for creating numbers and expressions for processing them. In addition, Python provides more advanced numeric programming support and objects for more advanced needs. A fairly complete inventory of Python's numeric toolbox includes:

- Integer and floating-point objects
- Complex number objects
- Decimal fixed-precision objects
- Fraction rational number objects
- Set objects and operations
- Boolean and bitwise operations
- Built-in modules, such as `math`, `cmath`, `random`, and `statistics`
- Third-party add-ons, including vectors, visualization, plotting, and extended precision

Because the object types in this list's first bullet item tend to see the most action in Python code, this chapter starts with basic numbers and fundamentals, then moves on to explore other types on this list, which serve specialized roles. We'll also study *sets* here, which have both numeric and collection qualities, but are generally considered more the former than the latter. Before we jump into code, though, the next few sections get us started with a brief overview of how we write and process numbers in our scripts.

Numeric Literals

Among its basic object types, Python provides *integers*, which are positive and negative whole numbers, and *floating-point* numbers, which are numbers with a fractional part (sometimes called *floats* for verbal economy). Python also allows us to write integers using hexadecimal, octal, and binary literals; offers a complex number type; and allows integers to have unlimited *precision*—they can grow to have as many digits as your memory space allows. [Table 5-1](#) shows what Python's numeric types look like when written out in a program as literals or constructor-function calls.

Table 5-1. Numeric literals and constructors

Literal	Interpretation
1234, -24, 0, 9_999_999_999_999	Integers (unlimited size)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Floating-point numbers
0o177, 0x9ff, 0b101010	Octal, hex, and binary literals
3+4j, 3.0+4.0j, 3j	Complex number literals
set('hack'), {1, 2, 3, 4}	Sets: constructors and literals
Decimal('1.0'), Fraction(1, 3)	Decimal and fraction extension types
bool(X), True, False	Boolean type and constants

In general, Python's numeric type literals are straightforward to write, but a few coding concepts are worth highlighting up front:

Integer and floating-point literals

Integers are written as strings of decimal digits. As noted, they have precision (number of digits) limited only by your device's available memory. You can easily compute 2 raised to the power 1,000,000, though with 300K digits, it may take some time to print (and can't be converted to a print string today by default, per [Chapter 4](#)).

Floating-point numbers have a decimal point and/or an optional signed exponent introduced by an e or E and followed by an optional sign. If you write a number with a decimal point or exponent, Python makes it a floating-point object and uses floating-point (not integer) math when the object is used in an expression. As you'll learn, mixing a floating-point number with an integer does floating-point math too, after converting the integer up.

Hexadecimal, octal, and binary literals

Integers may be coded in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2), the last three of which are common in some programming domains. Hexadecimals start with a leading 0x or 0X, followed by a string of hexadecimal digits (0–9 and A–F). Hex digits may be coded in lowercase or uppercase. Octal literals start with a leading 0o or 0O (zero and lowercase or

uppercase letter o), followed by a string of octal digits (0–7). Binary literals begin with a leading 0b or 0B, followed by binary digits (0–1).

All of these literals produce integer objects in program code; they are just alternative syntaxes for specifying values. The built-in calls `hex(I)`, `oct(I)`, and `bin(I)` convert an integer to its representation string in these three bases, and `int(str, base)` converts a runtime string to an integer per a given base.

Complex numbers

Though used more rarely, Python complex literals are written as *realpart+imaginarypart*, where the *imaginarypart* is terminated with a j or J. The *realpart* is technically optional, so the *imaginarypart* may appear on its own. Internally, complex numbers are implemented as pairs of floating-point numbers, but all numeric operations perform complex math when applied to complex numbers. Complex numbers may also be created with the `complex(real, imag)` built-in call.

Coding other numeric types

As you'll see later in this chapter, there are additional numeric types near the end of [Table 5-1](#) that serve more advanced or specialized roles. You create some of these by calling functions in imported modules (e.g., decimals and fractions), others have literal syntax all their own (e.g., sets), and Booleans are a kind of specialized integer.

Built-in Numeric Tools

Besides the built-in number literals and construction calls shown in [Table 5-1](#), Python provides a set of tools for processing number objects:

Expression operators

+, -, *, /, >>, **, &, %, etc.

Built-in mathematical functions

pow, abs, round, int, hex, bin, etc.

Utility modules

random, math, statistics, etc.

You'll meet all of these as we go along.

Although numbers are primarily processed with expressions, built-ins, and modules, they also have a handful of type-specific *methods* today, which you'll meet in this chapter. Floating-point numbers, for example, have an `as_integer_ratio` method useful for the fraction number type, and an `is_integer` method to test if the number is an integer. Integer attributes include a `bit_length` method that gives the number of bits necessary to represent the object's value, and as part collection and part number, *sets* support both expressions and methods too.

Since expressions are the most essential tool for most number types, though, let's turn to them next.

Python Expression Operators

The most fundamental tool that processes numbers is the *expression*: a combination of numbers (or other objects) and operators that computes a value when executed by Python. In Python, you write expressions using the usual mathematical notation and operator symbols. For instance, to add two

numbers *X* and *Y* you would say *X* + *Y*, which tells Python to apply the + operator to the values named by *X* and *Y*. The result of the expression is the sum of *X* and *Y*, another number object.

Table 5-2 lists all the operator expressions available in Python, abstractly. Many are self-explanatory; for instance, the usual mathematical operators (+, −, *, /, and so on) are supported. A few will be familiar if you’ve used other languages in the past: % computes a division remainder, << performs a bitwise left-shift, & computes a bitwise AND result, and so on. Others are more Python specific, and not all are numeric in nature: for example, the *is* operator tests object identity (i.e., same address in memory, a strict form of equality), and *lambda* creates unnamed functions.

Table 5-2. Python expression operators, by increasing precedence (binding)

Operators	Description
<code>yield x, yield from x</code>	Generator function send protocol
<code>x := y</code>	Assignment expression
<code>lambda args: expression</code>	Anonymous function generation
<code>x if y else z</code>	Ternary selection (<i>x</i> is evaluated only if <i>y</i> is true)
<code>x or y</code>	Logical OR (<i>y</i> is evaluated only if <i>x</i> is false)
<code>x and y</code>	Logical AND (<i>y</i> is evaluated only if <i>x</i> is true)
<code>not x</code>	Logical negation
<code>x in y, x not in y</code>	Membership (iterables)
<code>x is y, x is not y</code>	Object identity tests
<code>x < y, x <= y, x > y, x >= y</code>	Magnitude comparison, set subset and superset
<code>x == y, x != y</code>	Value equality operators
<code>x y</code>	Bitwise OR, set union, dictionary merge
<code>x ^ y</code>	Bitwise XOR, set symmetric difference
<code>x & y</code>	Bitwise AND, set intersection
<code>x << y, x >> y</code>	Shift <i>x</i> left or right by <i>y</i> bits
<code>x + y</code>	Addition, concatenation
<code>x - y</code>	Subtraction, set difference
<code>x * y</code>	Multiplication, repetition
<code>x % y</code>	Remainder, format
<code>x / y, x // y</code>	Division: true and floor
<code>x @ y</code>	Matrix multiplication (unused by Python)
<code>-x, +x, ~x</code>	Negation, identity Bitwise NOT (inversion)
<code>x ** y</code>	Power (exponentiation)
<code>await x</code>	Await expression (async functions)
<code>x[i]</code>	Indexing (sequence, mapping, others)
<code>x[i:j:k]</code>	Slicing
<code>x(...)</code>	Call (function, method, class, other callable)
<code>x.attr</code>	Attribute reference
<code>(...)</code>	Tuple, expression, generator expression
<code>[...]</code>	List, list comprehension
<code>{...}</code>	Dictionary, set, dictionary and set comprehensions

While [Table 5-2](#) works as a reference, some of its operators won't make sense until you've seen them in action, and some are more subtle than the table may imply. For instance:

- Parentheses are required for `yield` if it's not alone on the right side of an assignment statement, as well as the `:=` named-assignment operator if it's used in some contexts.
- Comparison operators compare all parts of collections automatically and may be chained as a shorthand and potential optimization (e.g., `X < Y < Z` produces the same result as `X < Y` and `Y < Z`).
- Python defines an `@` operator meant for matrix multiplication but does not provide an implementation for it; unless you code one in a class or use a library that does, this operator does *nothing*.
- The parentheses used for tuples, expressions, and generators may sometimes be omitted; when omitted for tuples, the *comma* separating its items acts like a lowest-precedence operator if not otherwise significant.
- Some operators, like `yield`, `lambda`, and `await`, have to do with larger topics that have little to do with numbers and can safely be ignored at this early point in your Python career.

This book will defer to Python's manuals for other minutiae, but you'll see most of the operators in [Table 5-2](#) in action later. First, though, we need to take a quick look at the ways these operators may be combined in expressions.



Meet the Python no-ops: The `@` character is used by Python to introduce function decorators (covered later in this book), but not as an expression operator—despite its being specified as such. In the latter role, `@` joins *Ellipsis* (`...`) and *type hinting* as tools defined but wholly unused by Python itself. As if you didn't have enough to learn with the real stuff!

Mixed Operators: Precedence

As in most languages, in Python, you code more complex expressions by stringing together the operator expressions in [Table 5-2](#). For instance, the sum of two multiplications might be written as a mix of variables and operators:

```
A * B + C * D
```

Which raises the question: how does Python know which operation to perform first? The answer to this question lies in *operator precedence*. When you write an expression with more than one operator, Python groups its parts according to what are called *precedence rules*, and this grouping determines the order in which the expression's parts are computed. To denote this, [Table 5-2](#) is ordered by operator precedence:

- Operators *lower* in the table have higher precedence, and so bind more *tightly* in mixed expressions. Put another way, operators *higher* in the table have lower precedence and bind less tightly than those below them.
- Operators in the *same row* in the table generally group from *left to right* when combined (except for exponentiation, which groups right to left, and comparisons, which chain left to right).

So, for example, if you write `X + Y * Z`, Python evaluates the multiplication first (`Y * Z`) then adds that result to `X`, because `*` has higher precedence (is lower in the table) than `+`. Similarly, in this section's

original example, both multiplications ($A * B$ and $C * D$) will happen before their results are added because $+$ is above $*$.

Parentheses Group Subexpressions

You can largely forget about precedence rules if you're careful to group parts of expressions with parentheses. When you enclose subexpressions in parentheses, you override Python's precedence rules; Python always evaluates expressions in parentheses *first* before using their results in the enclosing expressions.

For instance, instead of coding $X + Y * Z$, you could write one of the following to force Python to evaluate the expression in either desired order:

```
(X + Y) * Z
X + (Y * Z)
```

In the first case, $+$ is applied to X and Y first, because this subexpression is wrapped in parentheses. In the second case, the $*$ is performed first (just as if there were no parentheses at all). Generally speaking, adding parentheses in large expressions is a good idea—it not only forces the evaluation order you want, but also aids readability.

Mixed Types Are Converted Up

Besides mixing operators in expressions, you can also mix numeric types. For instance, you can add an integer to a floating-point number:

```
40 + 3.14
```

But this leads to another question: what type is the result—integer or floating point? The answer is simple, especially if you've used almost any other language before: in mixed-type numeric expressions, *operands* (the parts of the expression that aren't operators) are first converted *up* to the type of the most complicated operand, and then the math is performed on same-type operands. The result is that of the up-converted operands.

For this, Python ranks the complexity of numeric types like so: integers are simpler than floating-point numbers, which are simpler than complex numbers. So, when an integer is mixed with a floating point, as in the preceding example, the integer is converted up to a floating-point value first, and floating-point math yields the floating-point result. See for yourself in your local Python REPL:

```
>>> 40 + 3.14      # Integer to float, float math/result
43.14
```

Similarly, any mixed-type expression where one operand is a complex number results in the other operand being converted up to a complex number, and the expression yields a complex result. Conversions also run in equality and magnitude comparisons: $3 == 3.0$ is true, but $3 > 3.0$ is not.

You can force the issue by calling built-in functions to convert types manually:

```
>>> int(3.1415)    # Truncates float to integer
3
>>> float(3)       # Converts integer to float
3.0
```

However, you won't usually need to do this: because Python automatically converts up to the more complex type within an expression, the results are normally what you want.

While automatic conversions are run for both numeric and comparison operators, keep in mind that they apply only when mixing *numeric* objects (e.g., an integer and a float) in an expression. In general, Python does *not* convert across any other type boundaries automatically. Adding a string of digits to an integer, for example, results in an error, unless you manually convert one or the other; watch for an example and rationale when we explore strings in [Chapter 7](#). Equality tests do work on mixed types (e.g., a string is never equal to any integer), but magnitude comparisons do not.

Preview: Operator Overloading and Polymorphism

Although we're focusing on built-in numbers right now, all Python operators may be *overloaded* (i.e., implemented) by Python classes and C extension types to work on objects you create. For instance, you'll see later that objects coded with classes may be added or concatenated with `x+y` expressions, indexed with `x[i]` expressions, and so on.

Furthermore, Python itself automatically overloads some operators, such that they perform different actions depending on the type of built-in objects being processed. For example, the `+` operator performs addition when applied to numbers but performs concatenation when applied to sequence objects like strings and lists. In fact, `+` can mean anything at all when applied to objects you define with classes.

As we saw in the prior chapter, this property is usually called *polymorphism*—a term indicating that the meaning of an operation depends on the type of the objects being processed. We'll revisit this concept when we explore functions in [Chapter 16](#), because it becomes a much more obvious feature in that context.

Numbers in Action

On to the code! Probably the best way to understand numeric objects and expressions is to see them in action, so with all the preceding basics in hand, let's start up the interactive command line and try some simple but illustrative operations (be sure to see [Chapter 3](#) for pointers if you need help starting a REPL).

Variables and Basic Expressions

First of all, let's do the math to demo some basics. In the following interaction, we first assign two variables (`a` and `b`) to integers so we can use them later in a larger expression. *Variables* are simply names—created by you or Python—that are used to keep track of information in your program. We'll say more about this in the next chapter, but in Python:

- Variables are created when they are first assigned values.
- Variables are replaced with their values when used in expressions.
- Variables must be assigned before they can be used in expressions.
- Variables refer to objects and need not be declared ahead of time.

In other words, these assignments cause the variables `a` and `b` to spring into existence automatically:

```
$ python3                # Fire up a REPL
>>> a = 3                # Name created: no need to declare ahead of time
>>> b = 4
```

This code also uses *comments*. Recall from [Chapter 3](#) that in Python code, text after a `#` mark and continuing to the end of the line is considered to be a comment and is ignored by Python. Comments are

one way to write human-readable documentation for your code, and an important part of programming. They describe aspects of the code, salient or subtle, as an aid for others (and you, six months down the road!). In the next part of the book, you'll also meet a related feature—*documentation strings*—that attaches docs to objects so it's available after your code is loaded.

Again, though, because code you type interactively is temporary, you won't normally write comments in this context. If you're working along, this means you don't need to type any of the comment text from the `#` through to the end of the line in a REPL; it's not a required part of the statements we're running this way.

Now, let's use our new integer objects in expressions. At this point, the values of `a` and `b` are still 3 and 4, respectively. Variables like these are replaced with their values whenever they're used inside an expression, and the expression results are echoed back immediately and automatically when we're working interactively:

```
>>> a + 1, a - 1          # Addition (3 + 1), subtraction (3 - 1)
(4, 2)
>>> b * 3, b / 2          # Multiplication (4 * 3), division (4 / 2)
(12, 2.0)
>>> a % 2, b ** 2         # Modulus (remainder), power (4 ** 2)
(1, 16)
>>> 2 + 4.0, 2.0 ** b     # Mixed-type conversions
(6.0, 16.0)
```

Per [Chapter 3](#), the results being echoed back here are *tuples* of two values because the lines typed at the prompt contain two expressions separated by commas; that's why the results are displayed in parentheses. More importantly, these expressions work because the variables `a` and `b` within them have been assigned values. If you use a different variable that has *not yet been assigned*, Python reports an error rather than filling in some default value:

```
>>> c * 2
NameError: name 'c' is not defined
```

As also previewed in [Chapter 3](#), you don't need to predeclare variables in Python, but they must have been assigned at least once before you can use them. In practice, this means you have to initialize counters to zero before you can add to them, initialize lists to an empty list before you can append to them, and so on.

Here are two slightly larger expressions to illustrate operator grouping and more about conversions:

```
>>> b / 2 + a              # Same as ((4 / 2) + 3)
5.0
>>> b / (2 + a)           # Same as (4 / (2 + 3))
0.8
```

In the *first* expression, there are no parentheses, so Python automatically groups the components according to its precedence rules—because `/` is lower in [Table 5-2](#) than `+`, it binds more tightly and so is evaluated first. The result is as if the expression had been organized with parentheses as shown in the comment to the right of the code. In the *second* expression, parentheses are added around the `+` part to force Python to evaluate it first (i.e., before the `/`).

Also, notice that all the numbers are integers in each of these examples. Python's `/` performs *true* division, which always retains fractional remainders and gives a floating-point result—which is in turn reflected in the result of the whole expression. You can force a fractional result by coding `2.0` instead of `2` but don't have to. You can also opt to use *floor* division by coding these examples with `//` instead of `/`,

and Python will discard decimal digits in the result; because results reflect the types of operands, you'll get back truncated floating-point for floats:

```
>>> a, b                # Same, original values
(3, 4)

>>> b // 2 + a          # Floor division: integer
5
>>> b // (2 + a)        # Truncates fraction (for positives)
0

>>> b // 2.0 + a        # Auto-conversions: floating-point
5.0
>>> b // (2.0 + a)
0.0
```

You'll learn more about division later in this section.

Numeric Display Formats

Once you start playing with Python numbers in earnest, the results of some expressions may look a bit odd the first time you see them:

```
>>> 1.1 + 2.2           # What's up with the 3 at the end?
3.3000000000000003
>>> print(1.1 + 2.2)    # Same for prints
3.3000000000000003
```

The full story behind this odd result has to do with the limitations of floating-point hardware and its inability to exactly represent some values in a limited number of bits. Python floating-point numbers map to the underlying chips on your device and are only as accurate as those chips allow—a physical constraint that can be addressed with add-ons that extend floating-point precision, as well as techniques discussed in [“Floating-point equality” on page 93](#).

Because computer architecture is well beyond this book's scope, though, we'll finesse this by saying that your computer's floating-point hardware is doing the best it can, and neither it nor Python is in error here. In fact, this is partly a *display* issue—Python's floating-point display logic tries to be intelligent and usually shows fewer decimal digits, but occasionally cannot. Our earlier examples gave fewer digits automatically, and you can always force the issue in programs with string formatting:

```
>>> num = 1.1 + 2.2
>>> num                # Auto-echoes (and prints)
3.3000000000000003

>>> '%e' % num         # String-formatting expression
'3.300000e+00'
>>> '%.1f' % num       # Alternative floating-point format
'3.3'

>>> f'{num:e}', f'{num:.1f}' # F-strings (see also format method)
('3.300000e+00', '3.3')
```

The last three tests here employ flexible string formatting, which we will explore in full in the upcoming chapter on strings ([Chapter 7](#)). Its results are strings that are typically, but not always, printed to displays or reports.

Display Formats: str and repr

Although it's not yet obvious in this chapter, the default output format of interactive echoes and `print` technically correspond to the built-in `repr` and `str` functions, respectively:

```
>>> repr('hack')           # Used by echoes: as-code form
'"hack"'
>>> str('hack')            # Used by print: user-friendly form
'hack'
```

Both of these convert arbitrary objects to their string representations: `repr` (and the default interactive echo) produces results that look as though they were code; `str` (and the `print` operation) converts to a typically more user-friendly format if available. Some objects have both—a `str` for general use, and a `repr` with extra details. This notion will resurface when we explore both strings and operator overloading in classes.

Besides providing print strings for arbitrary objects, the `str` built-in is also the name of the string object type, which can be called with an encoding name to decode a Unicode string from a byte string as an alternative to the `bytes.decode` method introduced briefly in [Chapter 4](#). You'll learn more about this advanced `str` role in [Chapter 37](#).

Comparison Operators

So far, we've been dealing with standard numeric operations (e.g., addition and multiplication), but numbers, like all Python objects, can also be compared. Normal comparisons work for numbers exactly as you'd expect—they compare the relative magnitudes of their operands and return a Boolean result, which we would normally test and take action on in a larger statement and program (e.g., see the intro to `if` in [Chapter 4](#)):

```
>>> 1 < 2                   # Less than (magnitude)
True
>>> 2.0 >= 1                # Greater than or equal: mixed-type 1 converted to 1.0
True
>>> 2.0 == 2.0              # Equal value
True
>>> 2.0 != 2.0              # Not equal value
False
```

Notice again how mixed types are allowed in numeric expressions (only); in the second test here, Python compares values in terms of the more complex type, float.

Chained comparisons

Interestingly, Python also allows us to *chain* multiple comparisons together to perform range tests. Chained comparisons are a sort of shorthand for larger Boolean expressions. In short, Python lets us string together magnitude comparison tests to code chained comparisons such as range tests. The expression `(A < B < C)`, for instance, tests whether B is between A and C, noninclusively; it is equivalent to the Boolean test `(A < B and B < C)` but is easier on the eyes (and the keyboard). For example, assume the following assignments:

```
>>> X = 2
>>> Y = 4
>>> Z = 6
```

The following two expressions have identical effects, but the first is shorter to type, and it may run slightly faster since Python needs to evaluate *Y* only once—and it may matter if *Y* is a call to a complicated function:

```
>>> X < Y < Z          # Chained comparisons: range tests
True
>>> X < Y and Y < Z
True
```

The same equivalence holds for false results, and arbitrary chain lengths are allowed:

```
>>> X < Y > Z
False
>>> X < Y and Y > Z
False

>>> 1 < 2 < 3.0 < 4
True
>>> 1 > 2 > 3.0 > 4
False
```

You can use other comparisons in chained tests, but the resulting expressions can become nonintuitive unless you evaluate them the way Python does. The first of the following, for instance, is false just because 1 is not equal to 2:

```
>>> 1 == 2 < 3          # Same as: (1 == 2) and (2 < 3), but not: False < 3!
False
>>> True is False is True # Same as: (True is False) and (False is True)
False
```

Python does not compare the `1 == 2` expression's `False` result to 3—this would technically mean the same as `0 < 3`, which would be `True` (you'll learn more about the `True` and `False` objects later in this chapter and explore the rarely used `is` identity operator in the next).

Floating-point equality

One last note here before we move on: chaining aside, numeric comparisons are based on magnitudes, which are generally simple—though *floating-point* numbers may not always work as you'd expect, and may require conversions or other massaging to be compared meaningfully:

```
>>> 1.1 + 2.2 == 3.3      # Shouldn't this be True?
False
>>> 1.1 + 2.2             # Close to 3.3, but not exactly: limited precision
3.3000000000000003
```

This is related to the earlier coverage of numeric display formats and stems from the fact that floating-point numbers cannot represent some values exactly due to their limited number of bits—a fundamental issue in numeric programming not unique to Python. To accommodate this imprecision in equality tests, either truncate, round, use floors, or, as of Python 3.5, import and call the `math` standard-library module's `isclose`, which is true if values are within a tolerance of each other (there's more on `math` and floors ahead, and more on `isclose` in Python's manuals):

```
>>> int(1.1 + 2.2) == int(3.3)      # OK if convert: see also floor, trunc ahead
True
>>> round(1.1 + 2.2, 1) == round(3.3, 1)
True
>>> import math                      # Import modules to call their functions
>>> math.isclose(1.1 + 2.2, 3.3)     # Within default-but-passable tolerances
True
```

We'll revisit this later in this chapter when we meet *decimals* and *fractions*, which can also address such limitations. First, though, let's continue our tour of Python's core numeric operations, with a deeper look at division.

Division Operators

Python has two division operators introduced earlier, as well as one that's strongly related. Here's the whole gang:

X / Y

Called *true* division, this always keeps remainders in floating-point results, regardless of types.

$X // Y$

Called *floor* division, this always truncates fractional remainders down to their floor, regardless of types, and its result type depends on the types of its operands.

$X \% Y$

Called *modulus*, this returns a division's remainder, with a result type that varies per operand types. This also does formatting when used on strings, per [Chapter 7](#).

The following demos the two *division* operators at work:

```
>>> 10 / 4                          # True div: keeps remainder always
2.5
>>> 10 / 4.0                        # Same for floats
2.5
>>> 10 // 4                         # Floor div: drops remainder always
2
>>> 10 // 4.0                       # Same for floats, but type varies
2.0
```

Notice that the object type of the result for `//` is dependent on its operand's types: if either is a float, the result is a float; otherwise, it is an integer. If you want to ensure an integer result, simply wrap the expression in `int` to convert:

```
>>> int(10 // 4.0)
2
```

The related *modulus* returns the remainder of division with a type to match operands (useful when your code needs to know how much is “left over” after a `//`), and the `divmod` built-in function gives both parts when needed:

```
>>> 10 % 3, 10 % 3.0                # Remainder of division: (3 * 3) + 1
(1, 1.0)
>>> divmod(10, 3)                   # Both parts of division in a tuple
(3, 1)
```


Floor versus truncation

One subtlety here: the `//` operator is informally called *truncating* division, but it's more accurate to refer to it as *floor* division—it truncates the result down to its floor, which means the closest whole number below the true result. The net effect is to round down, not strictly truncate, and this matters for negatives. You can see the difference for yourself with the Python `math` module (as you've learned, modules must be imported before you can use their contents):

```
>>> import math
>>> math.floor(2.5)          # Closest number below value
2
>>> math.floor(-2.5)         # But not truncation for negative!
-3
>>> math.trunc(2.5)          # Truncate fractional part (toward zero)
2
>>> math.trunc(-2.5)         # And is truncation for negative
-2
```

When running division operators, you only really truncate for positive results, since truncation is then the same as floor; for negatives, it's a floor result. Really, they are both floor, but floor just happens to be the same as truncation for positives (*cut-and-pasters*: if minus signs morph to Unicode dashes in this book, replace them with simple ASCII “-” hyphens to run; Python requires the latter for the minus sign, and tools are notorious for botching this):

```
>>> 5 / 2, 5 / -2            # True division keeps remainders
(2.5, -2.5)

>>> 5 // 2, 5 // -2          # Truncates to floor: rounds to first lower integer
(2, -3)

>>> 5 / 2.0, 5 / -2.0        # Ditto for floats
(2.5, -2.5)

>>> 5 // 2.0, 5 // -2.0      # Though result is float too
(2.0, -3.0)
```

If you really want truncation toward zero regardless of sign, you can always run a true division result through `math.trunc` (as demoed earlier, the `round` built-in has related functionality and the `int` built-in has the same effect, and neither requires an import):

```
>>> import math
>>> 5 / -2                    # Keep remainder
-2.5
>>> 5 // -2                   # Floor below result
-3
>>> math.trunc(5 / -2)        # Truncate instead of floor (same as int())
-2
```

So why the fuss over truncation? This won't be obvious until you graduate to writing larger Python programs later in this book, but it's an essential tool in some use cases. Watch for a prime-number `while` loop example in [Chapter 13](#) and a corresponding exercise at the end of [Part IV](#) that wholly rely on the truncating behavior of `//`.

convenient alternatives for specifying values, especially when their mapping to bytes and bits is important. We detailed coding rules before; let's try these out live.

As noted earlier, these other-base *literals* are simply an alternative syntax for specifying the value of an integer object. For example, the following literals produce normal integers with the specified values. In memory, an integer's value is the same, regardless of the base we use to specify it in our code:

```
>>> 0x01, 0x10, 0xFF          # Hex literals: base 16, digits 0-9/A-F
(1, 16, 255)
>>> 0o1, 0o20, 0o377          # Octal literals: base 8, digits 0-7
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111  # Binary literals: base 2, digits 0-1
(1, 16, 255)
```

Here, the hex value 0xFF, the octal value 0o377, and the binary value 0b11111111 are all decimal 255. The F digits in the hex value, for example, each mean 15 in decimal and a 4-bit 1111 in binary, and reflect powers of 16. Thus, the hex value 0xFF and others convert to decimal values as follows:

```
>>> 0xFF, (15 * (16 ** 1)) + (15 * (16 ** 0))    # How hex/binary map to decimal
(255, 255)
>>> 0x2F, (2 * (16 ** 1)) + (15 * (16 ** 0))
(47, 47)
>>> 0xF, 0b1111, (1*(2**3) + 1*(2**2) + 1*(2**1) + 1*(2**0))
(15, 15, 15)
```

Python prints integer values in decimal (base 10) by default, but it also provides built-in functions that convert integers to other bases' digit strings formatted per Python-literal syntax—useful when programs or users expect to see values in a given base:

```
>>> oct(64), hex(64), bin(64)          # Numbers=>digit strings
('0o100', '0x40', '0b1000000')
```

The `oct` function converts decimal to octal, `hex` to hexadecimal, and `bin` to binary—all as strings. To go the other way, the built-in `int` function converts a string of digits to an integer, and an optional second argument lets you specify the numeric base—useful for numbers read from files as strings instead of coded in scripts:

```
>>> 64, 0o100, 0x40, 0b1000000        # Digits=>numbers in scripts and strings
(64, 64, 64, 64)

>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)

>>> int('0x40', 16), int('0b1000000', 2)  # Literal forms supported too
(64, 64)
```

The `eval` function can also be used to convert digit strings to numbers, because it treats strings as though they were Python code. Therefore, it has a similar effect, but usually runs more *slowly*—it actually compiles and runs the string as a piece of a program, and it assumes the string being run comes from a *trusted source*—a clever user might be able to submit a string that deletes files on your machine. In other words, be sparing and careful with this call:

```
>>> eval('64'), eval('0o100'), eval('0x40'), eval('0b1000000')
(64, 64, 64, 64)
```

Finally, you can also convert integers to base-specific strings with any of Python's three *string-formatting* tools, though you'll have to take this partly on faith until we reach strings' full coverage in [Chapter 7](#):

```
>>> '%o, %x, %#X' % (64, 255, 255)           # Numbers=>digits formatting*3
'100, ff, 0xFF'

>>> '{:o}, {:b}, {:x}, {:#X}'.format(64, 64, 255, 255)
'100, 1000000, ff, 0xFF'

>>> f'{64:o}, {64:b}, {255:x}, {255:#X}'        # The newest latest-and-greatest
'100, 1000000, ff, 0xFF'
```

In this code, o, b, and x format as octal, binary, and hex, respectively, and #X adds a base prefix and uses uppercase. As an aside, you can avoid the repeated inputs in each of these three formatting tools (but you're probably starting to see why picking just one is generally a good idea—and why feature redundancy is generally a bad idea!):

```
>>> '%(i)o, %(j)x, %(j)#X' % dict(i=64, j=255)
'100, ff, 0xFF'
>>> '{0:z}, {0:b}, {1:x}, {1:#X}'.format(64, 255)
'100, 1000000, ff, 0xFF'
>>> 'f'[{i:=64}:o], {i:=255}:x], {i:#X}'
'100, 1000000, ff, 0xFF'
```

Before we move on, keep in mind that other-base literals and converters support *arbitrarily* large integers too. The following, for instance, creates an integer in hex and displays it in decimal and octal and binary with converters:

[illegible]

Speaking of binary digits, the next section takes us on a tour of tools that process numbers' individual bits.

Bitwise Operations

Besides the normal numeric operations (addition, subtraction, and so on), Python supports most of the numeric expressions available in the C language. This includes operators that treat integers as strings of *binary bits* and can come in handy if your Python code must deal with things like network packets, serial ports, or packed binary data produced by or intended for a C program.

We can't dwell on the fundamentals of Boolean math here—again, those who must use it probably already know how it works, and others can often postpone the topic altogether—but the basics are straightforward. For instance, here are some of Python's bitwise expression operators at work performing bitwise shift and Boolean operations on integers:

```
>>> x = 1           # 1 decimal is 0001 in bits
>>> x << 2          # Shift left 2 bits: 0100
4
>>> x | 3           # Bitwise OR (either bit=1): 0001 | 0011
3
>>> x & 3           # Bitwise AND (both bits=1): 0001 & 0011
1
```

In the first expression, a binary 1 (in base 2, 0001) is shifted left two slots to create a binary 4 (0100). The last two operations perform a binary OR to combine bits (0001|0011 = 0011) and a binary AND to

select common bits ($0001 \& 0011 = 0001$). Such bit-masking operations allow us to encode and extract multiple flags and other values within a single integer.

This is one area where the binary and hexadecimal number support in Python become especially useful—they allow us to code and inspect numbers by bit-strings:

```
>>> X = 0b0001          # Binary literals
>>> X << 2              # Shift left
4
>>> bin(X << 2)         # Binary digits string
'0b100'

>>> bin(X | 0b0011)     # Bitwise OR: either
'0b11'
>>> bin(X & 0b11)       # Bitwise AND: both
'0b1'
```

This is also true for values that begin life as hex literals, or undergo base conversions:

```
>>> X = 0xFF           # Hex literals
>>> bin(X)
'0b11111111'
>>> X ^ 0b10101010     # Bitwise XOR: either but not both
85
>>> bin(X ^ 0b10101010)
'0b1010101'

>>> int('01010101', 2) # Digits=>number: string to int per base
85
>>> hex(85)            # Number=>digits: Hex digit string
'0x55'
```

Also in this department, Python integers come with a `bit_length` method, which allows you to query the number of bits required to represent the number’s value in binary. Sharp-eyed readers might point out that you can often achieve the same effect by subtracting 2 from the length of the `bin` string using the `len` built-in function we first used in [Chapter 4](#) (to account for the leading “0b”), though its temporary string result may make it less efficient:

```
>>> X = 99
>>> bin(X), X.bit_length(), len(bin(X)) - 2
('0b1100011', 7, 7)
>>> bin(256), (256).bit_length(), len(bin(256)) - 2
('0b100000000', 9, 9)
```

We won’t go into much more detail on such “bit twiddling” here. It’s supported if you need it, but bit-wise operations are often not as important in a high-level language such as Python as they are in a low-level language such as C. As a rule of thumb, if you find yourself wanting to flip bits in Python, you should think about which language you’re really coding. As you’ll see in upcoming chapters, Python’s lists, dictionaries, and the like provide richer ways to encode information than bit strings, especially when your data’s audience includes readers of the human variety.

Underscore Separators in Numbers

If you’re finding it hard to read the longer digit strings in this chapter, there’s some good news: as of 3.6, numeric literals in Python can be coded with embedded underscores (“_”) to group digits for easier viewing. These underscores don’t modify number values; Python simply discards them after reading your code. They do, however, work on all the numbers and bases we’ve met, including complex-number

parts and float-point decimal digits, and can enhance the readability of numeric literals in your scripts. Here's how they look—with before on the left and after on the right:

```
>>> 999999999999 == 9_999_999_999_999          # Decimal: thousands
True
>>> 0xFFFFFFFF == 0xFF_FF_FF_FF                 # Hex: 8-bit bytes
True
>>> 0o777777777777 == 0o777_777_777_777         # Octal: 9 bits each
True
>>> 0b1111111111111111 == 0b1111_1111_1111_1111 # Binary: 4-bit nibbles
True
>>> 3.141592653589793 == 3.141_592_653_589_793   # Float: decimal digits
True
>>> 123456789.123456789 == 123_456_789.123_456_789 # Float: both sides
True
```

While this is a useful feature, you should keep in mind that it's just skin deep. For example, numbers lose their underscores once read. You can add back comma and underscore separators with the *string-formatting* method and f-string we'll explore in [Chapter 7](#), but this is mostly just for display, and the originals are lost:

```
>>> x = 9_999_998                                # Your number with "_"s for digit groupings
>>> x                                              # But dropped when read: not in displays
99999998
>>> x + 1                                         # Ditto for derived computation results
99999999

>>> f'{x:,.} and {x:._}'                        # Formatting adds separators, but just for show
'9,999,998 and 9_999_998'
```

Moreover, Python doesn't do any sort of sanity checks on underscores, except for disallowing leading, trailing, and multiple-appearance uses. The underscores are really just digit “spacers” that can be used—and misused—arbitrarily:

```
>>> 99_9                                          # No position-error checking provided
999
>>> 1_23_456_7890                               # Hmm...
1234567890

>>> _9
NameError: name '_9' is not defined. Did you mean: '_'?
>>> 9_
SyntaxError: invalid decimal literal
>>> 9_9_9
SyntaxError: invalid decimal literal

>>> 9_9_9                                         # Syntax oddities checked, semantics not
999
>>> hex(0xf_ff_fff_f_f)                         # And Python won't retain your "_"s
'0xffffffff'
```

Also bear in mind that *commas* cannot be used in numeric literals you code—despite their similarity, underscores are essentially ignored as documentation, but commas are taken to be *tuple* item separators if erroneously used:

```
>>> 12_345_678                                  # Underscores are ignored
12345678
>>> 12,345,678                                  # But commas mean a tuple!
(12, 345, 678)
```

Underscores can enhance readability to be sure, but they are largely cosmetic and apply only to large numeric literals in your code—because typical Python programs *compute* most numbers rather than *hardcoding* them, this seems likely to be uncommon in practice. A more promising use case is *input*—string-to-number converters allow underscores too:

```
>>> int('1_234_567')           # Works in text read from data files too
1234567
>>> eval('1_234_567')          # But does raw-data readability matter?
1234567
>>> float('1_2_34.567_8_90')
1234.56789
```

But it's difficult to justify underscores on data alone, given that scripts could simply strip underscores themselves. Like all such tools, use when it makes sense (and don't be shocked if this crops up in unfair interview questions!).

Other Built-in Numeric Tools

In addition to its core object types, Python also provides both built-in *functions* and standard-library *modules* for numeric processing. The `pow` and `abs` built-in functions, for instance, compute powers and absolute values, respectively. Here's a brief roundup of common tools in the built-in `math` module (which contains most of the tools in the C language's math library), along with a few numeric built-in functions:

```
>>> import math
>>> math.pi, math.e              # Common constants
(3.141592653589793, 2.718281828459045)

>>> math.sin(2 * math.pi / 180)  # Sine, tangent, cosine
0.03489949670250097

>>> math.sqrt(144), math.sqrt(2)  # Square root
(12.0, 1.4142135623730951)

>>> pow(2, 4), 2 ** 4, 2.0 ** 4.0  # Exponentiation (power)
(16, 16, 16.0)

>>> abs(-62.0), sum((1, 2, 3, 4))  # Absolute value, summation
(62.0, 10)

>>> min(3, 1, 2, 4), max(3, 1, 2, 4) # Minimum, maximum
(1, 4)
```

The `sum` function shown here works on a sequence (really, *iterable*) of numbers, and `min` and `max` accept either a collection or individual arguments. There are also multiple ways to drop the decimal digits of floating-point numbers, both for calculations and displays; some of these are richer than previously shown:

```
>>> math.floor(2.567), math.floor(-2.567)  # Floor: next-lower integer
(2, -3)

>>> math.trunc(2.567), math.trunc(-2.567)  # Truncate: drop digits
(2, -2)

>>> int(2.567), int(-2.567)                # Truncate: alternative
(2, -2)

>>> round(2.567), round(2.567, 2), round(2567, -3) # Round to digits (+/-)
```

```
(3, 2.57, 3000)
```

```
>>> '%.1f' % 2.567, '{0:.2f}'.format(2.567)      # Format display (Chapter 7)
('2.6', '2.57')
```

As shown earlier, the last of these produces strings that we would usually print and supports a variety of formatting options. String formatting is still subtly different, though: `round` rounds and drops decimal digits but still produces a number in memory, whereas string formatting produces a string, not a number:

```
>>> (1 / 3.0), round(1 / 3.0, 2), f'{(1 / 3.0):.2f}'
(0.3333333333333333, 0.33, '0.33')
```

Interestingly, there are three ways to compute *square roots* in Python: using a module function, an expression, or a built-in function (if you're interested in performance, we will revisit these in an exercise and its solution at the end of [Part IV](#), to see which runs quicker):

```
>>> import math
>>> math.sqrt(144)          # Module
12.0
>>> 144 ** .5               # Expression
12.0
>>> pow(144, .5)           # Built-in
12.0
```

Notice that standard-library modules such as `math` must be imported, but built-in functions such as `abs` and `round` are always available without imports. This is because modules are external components, but built-in functions live in an implied namespace that Python automatically searches to find names used in your program. This namespace simply corresponds to the standard-library module called `builtins`, and there is much more about name resolution in the function and module parts of this book; for now, when you hear “module,” think “import.”

The standard library's `statistics` and `random` modules must be imported as well. Both modules provide an array of tools; `statistics` supports operations commonly found on calculators, and `random` enables tasks such as picking a random number between 0 and 1 and selecting a random integer between two numbers:

```
>>> import statistics
>>> statistics.mean([1, 2, 4, 5, 7])             # Average, median
3.8
>>> statistics.median([1, 2, 4, 5, 7])          # And a whole lot more: see its docs
4

>>> import random
>>> random.random()
0.5566014960423105
>>> random.random()                            # Random floats, integers, choices, shuffles
0.051308506597373515

>>> random.randint(1, 10)
5
>>> random.randint(1, 10)
9
```

The `random` module can also *choose* an item at random from a sequence, and *shuffle* a list of items randomly:


```
>>> random.choice(['Pizza', 'Tacos', 'Tikka', 'Lasagna'])
'Tikka'
>>> random.choice(['Pizza', 'Tacos', 'Tikka', 'Lasagna'])
'Lasagna'

>>> suits = ['hearts', 'clubs', 'diamonds', 'spades']
>>> random.shuffle(suits)
>>> suits
['spades', 'hearts', 'diamonds', 'clubs']
>>> random.shuffle(suits)
>>> suits
['clubs', 'diamonds', 'hearts', 'spades']
```

Though we'd need additional code to make this more tangible here, the `random` module can be useful for shuffling cards in games, picking images at random in a slideshow GUI, performing statistical simulations, and much more. We'll deploy it again later in this book (e.g., in [Chapter 20](#)'s permutations case study), but for more details, consult Python's library manual.

Other Numeric Objects

So far in this chapter, we've been using Python's core numeric types—integer, floating point, and complex. These will suffice for most of the number crunching that many programmers will ever need to do. Python comes with a handful of more exotic numeric types, though, that merit a brief look here.

Decimal Objects

First up, is Python's special-purpose numeric object known formally as `Decimal` (and informally as decimal). Syntactically, decimals are created by calling a function within an imported standard-library module, rather than running a literal expression. Functionally, decimals are like floating-point numbers, but they have a fixed and configurable number of decimal digits. Hence, decimals are *fixed-precision* floating-point values.

For example, with decimals, we can have a floating-point value that always retains just two decimal digits. Furthermore, we can specify how to round or truncate the extra decimal digits beyond the object's cutoff. Although it generally incurs a performance penalty compared to normal floating point, decimal is well suited to representing fixed-precision quantities like sums of money and can achieve better numeric accuracy in some contexts.

Decimal basics

As we learned when we explored comparisons, floating-point math is less than exact because of the limited space used to store values. For instance, the following should yield zero, but it does not. The result is close to zero, but there are not enough bits to be precise here:

```
>>> 0.1 + 0.1 + 0.1 - 0.3                                     # Almost zero, but not quite
5.551115123125783e-17
```

Using `print` for the user-friendly display format doesn't help here, because the hardware related to floating-point math is inherently limited in terms of accuracy (a.k.a. *precision*). With decimals, however, the result can be dead-on:

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

As shown here, we can make decimal objects by calling the `Decimal` constructor function in the `decimal` module and passing in strings that have the desired number of decimal digits for the resulting object (using the `str` function to convert floating-point values to strings if needed). When decimals of different precision are mixed in expressions, Python converts up to the largest number of decimal digits automatically:

```
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.1000') - Decimal('0.30')
Decimal('0.0000')
```

It's also possible to create a decimal object from a floating-point object, with either a call to `Decimal.from_float` or by passing floating-point numbers directly:

```
>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3)
Decimal('2.775557561565156540423631668E-17')
```

The conversion is exact but can yield a large default number of digits, unless they are fixed per the next section.

Setting decimal precision

Other tools in the `decimal` module can be used to set the precision of all decimal numbers, arrange error handling, and more. For instance, a context object in this module allows for specifying precision (number of decimal digits) and rounding modes (down, ceiling, etc.). The precision is applied globally for all decimals created by the caller:

```
>>> import decimal
>>> decimal.Decimal(1) / decimal.Decimal(7)                # Default precision
Decimal('0.1428571428571428571428571429')

>>> decimal.getcontext().prec = 4                           # Fixed precision
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')

>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3) # Closer to 0...
Decimal('1.110E-17')
```

Technically, significance is determined by digits input, and precision is applied on math operations. Although more subtle than we can explore in this brief overview, this property can make decimals useful as the basis for some monetary applications and may sometimes serve as an alternative to manual rounding and string formatting.

Because use of the decimal type is relatively rare in practice, though, this book will defer to Python's interactive `help` function and standard-library manuals for more details. And because decimals address some of the same floating-point accuracy issues as the fraction type, let's move on to the next section to see how the two compare.

Fraction Objects

Python's standard-library `fractions` module implements a *rational number* object. It essentially keeps both a numerator and a denominator explicitly, so as to avoid some of the inaccuracies and limitations of floating-point math. Like decimals, fractions do not map as closely to computer hardware as floating-point numbers. This means their performance may not be as good, but it also allows them to provide extra utility in a standard tool where useful.

Fraction basics

`Fraction` is a functional cousin to the `Decimal` fixed-precision object of the prior section, as both can be used to address the floating-point object's numerical inaccuracies. It's also used in similar ways—like `Decimal`, `Fraction` resides in a module; import its constructor and pass in a numerator and a denominator to make one (among other schemes). The following interaction shows how:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)           # Numerator, denominator
>>> y = Fraction(4, 6)          # Simplified to 2, 3 by gcd

>>> x
Fraction(1, 3)
>>> y
Fraction(2, 3)
>>> print(y)
2/3
```

Once created, `Fractions` can be used in mathematical expressions as usual:

```
>>> x + y
Fraction(1, 1)
>>> x - y           # Results are exact: numerator, denominator
Fraction(-1, 3)
>>> x * y
Fraction(2, 9)
```

`Fraction` objects can also be created from floating-point number strings, much like decimals:

```
>>> Fraction('.25')
Fraction(1, 4)
>>> Fraction('1.25')
Fraction(5, 4)

>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
```

Numeric accuracy in fractions and decimals

`Fraction` math is different from floating-point-type math, which is constrained by the underlying limitations of floating-point hardware. To compare, here are the same operations run with floating-point objects, and notes on their limited accuracy—they may display fewer digits in recent Pythons than they used to, but they still aren't exact values in memory:

```
>>> a = 1 / 3           # Only as accurate as floating-point hardware
>>> b = 4 / 6           # Can lose precision over many calculations
>>> a
0.3333333333333333
>>> b
0.6666666666666666

>>> a + b
1.0
>>> a - b
-0.3333333333333333
>>> a * b
0.2222222222222222
```

This floating-point limitation is especially apparent for values that cannot be represented accurately given their limited number of bits in memory. Both `Fraction` and `Decimal` provide ways to get exact

results, albeit at the cost of some lost speed and added code verbosity. For instance, in the following example (repeated from the prior section), floating-point numbers do not accurately give the zero answer expected, but both of the other types do:

```
>>> 0.1 + 0.1 + 0.1 - 0.3          # This should be zero (close, but not exact)
5.551115123125783e-17

>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)

>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

Moreover, fractions and decimals both allow more intuitive and accurate results than floating points sometimes can, in different ways—by using rational representation and by limiting precision:

```
>>> 1 / 3                          # Normal floating-point
0.3333333333333333

>>> Fraction(1, 3)                 # Numeric accuracy, two ways
Fraction(1, 3)

>>> import decimal
>>> decimal.getcontext().prec = 2
>>> Decimal(1) / Decimal(3)
Decimal('0.33')
```

In fact, fractions both retain accuracy and automatically simplify results. Continuing the preceding interaction:

```
>>> (1 / 3) + (6 / 12)
0.8333333333333333

>>> Fraction(1, 3) + Fraction(6, 12)
Fraction(5, 6)

>>> decimal.Decimal(1 / 3) + decimal.Decimal(6 / 12)
Decimal('0.83')
```

To support conversions, floating-point objects have an `as_integer_ratio` method noted earlier that yields numerator and denominator; fractions have a `from_float` method; and `float` accepts a `Fraction` as an argument. Because `Fraction` is also a lesser-used utility, though, we're going to stop short here too; for more details on `Fraction`, experiment further on your own and consult Python's documentation.

Set Objects

In addition to all the numeric objects we've explored, Python has built-in support for *sets*—an unordered collection of unique and immutable objects that supports operations corresponding to mathematical set theory. Sets straddle the fence between collections and math but lean far enough on the latter side to warrant coverage in this chapter.

By definition, an item appears only once in a set, no matter how many times it is added. Accordingly, sets have a variety of applications, especially in numeric and database-focused work. On the other hand, because sets are collections of other objects, they share some behavior with objects such as lists and

dictionaries previewed in [Chapter 4](#). For example, sets are iterable, can grow and shrink on demand, and may contain a variety of object types.

Still, because sets are unordered and do not map keys to values, they are neither sequence nor mapping types; they are a type category unto themselves. Moreover, because sets also tend to be used much less often than pervasive objects like lists and dictionaries, a brief look should suffice for most readers; let's get started here with the usual REPL tour.

Sets in action

First off, there are two ways to make sets—by call and literal. The *literal* uses the same “{ }” braces as dictionaries but simply enumerates items (there is no key) and allows you to initialize the set with individual objects. The *call* to `set` accepts an existing sequence (or other iterable) of items to add to the new set and is required to make an empty set (`{ }` is reserved for an empty dictionary). When sets are printed, they prefer the literal form, except when empty:

```
>>> x = set('abcde')           # Make a set by calling its type/function
>>> y = {99, 'b', 'y', 'd', 1.2} # Make a set by literal

>>> x
{'d', 'c', 'e', 'a', 'b'}      # Order is scrambled, displays literal
>>> y
{1.2, 99, 'y', 'd', 'b'}
```

Notice that sets don't maintain insertion order, unlike the keys in a dictionary (see [Chapter 4](#)'s introduction). This is by definition—sets are just groups of items—but the lack of positional ordering means that sequence operations won't work on sets. Per the following, empty sets require a call, a `*` in a literal unpacks items, both call and `*` accept any iterable, and sets always filter out duplicate entries; again, this is just how sets work in Python and elsewhere:

```
>>> z = set()                  # Make empty set: {} is empty dictionary
>>> z
set()

>>> z = set([1.2, 'a', 3, 1.2, 'a']) # Any sequence works, duplicates dropped
>>> z
{'a', 1.2, 3}

>>> {1, *'abc', *[1, 2, 3]}      # Literal star unpacking (Python 3.5+)
{1, 2, 3, 'c', 'b', 'a'}
```

Once you have sets, expression operators invoke set operations. Here are the most common in action; to run any of these on plain sequences like strings and list, you must first create a set of their items:

```
>>> x = set('abcd')
>>> y = set('bdxy')

>>> x - y                       # Difference: in x, not in y
{'a', 'c'}

>>> x | y                       # Union: in either
{'y', 'd', 'x', 'c', 'a', 'b'}

>>> x & y                       # Intersection: in both
{'d', 'b'}

>>> x ^ y                       # Symmetric difference: not in both
{'y', 'x', 'c', 'a'}
```

```
>>> x < y, x > y                # Superset, subset tests
(False, False)
```

An exception: the `in` set membership test expression is also defined to work on all other collection types, where it also performs membership (or a search, if you prefer to think in procedural terms). Hence, we do not need to convert things like strings and lists to sets to run this test:

```
>>> 'd' in x                      # Membership (sets)
True

>>> 'd' in 'code', 2 in [1, 2, 3]  # But works on other types too
(True, True)
```

In addition to expressions, the set object provides *methods* that correspond to these operations and more, and that support set changes. For instance, the set `add` method inserts one item, `update` is an in-place union, and `remove` deletes an item by value (per the prior chapter, run a `dir` call on any set instance or the set type name to see all the available methods). Assuming `x` and `y` are still as they were in the prior interaction:

```
>>> z = x.intersection(y)        # Same as x & y
>>> z
{'d', 'b'}
>>> z.add('HACK')                # Insert one item
>>> z
{'HACK', 'd', 'b'}
>>> z.update(set(['X', 'Y']))     # Merge: in-place union
>>> z
{'X', 'HACK', 'd', 'b', 'Y'}
>>> z.remove('b')                # Delete one item
>>> z
{'X', 'HACK', 'd', 'Y'}
```

Sets also are *iterable* (i.e., they support the iteration protocol introduced in the prior chapter) and hence can also be used in operations such as `len`, `for` loops, and list comprehensions. Because they are unordered, though, they don't support sequence operations like indexing, slicing, or concatenation:

```
>>> for item in set('abc'):       # See Chapter 4 for "for"
    print(item * 3)

aaa
ccc
bbb
>>> {'a', 'b', 'c'} + {'d'}
TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

Finally, although the set expressions shown earlier generally require two sets, their method-based counterparts can often work with *any iterable* as well—and may run faster because of it (though speed guesses are perilous in Python):

```
>>> S = set([1, 2, 3])
>>> S | set([3, 4])               # Expressions require both to be sets
{1, 2, 3, 4}
>>> S | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> S.union([3, 4])               # But their methods allow any iterable
{1, 2, 3, 4}
>>> S.intersection((1, 3, 5))
{1, 3}
```

```
>>> S.issubset(range(-5, 5))      # Subset of range -5...4 series generator
True

>>> S = set([1, 2, 3])
>>> S.intersection_update([1, 2, 5])
>>> S
{1, 2}
>>>
>>> S |= {1, 2, 4}
>>> S
{1, 2, 4}
```

For more details on set operations, see Python’s library manual. Among topics skipped here, sets also support in-place changes with additional methods, as well as assignment operators we’ll study later (e.g., `&=` and `|=`). Although sets can be coded manually in Python with other types like lists and dictionaries (and often were in the past), Python’s built-in sets use efficient algorithms and implementation techniques to provide quick and standard operations.

Immutable constraints and frozen sets

Sets are powerful and flexible objects, but they do have one constraint that you should keep in mind—largely because of their implementation, sets can contain only *immutable* object types (Python refers to this as “hashable,” which is close enough to “immutable” to use the latter here). Hence, lists and dictionaries cannot be embedded in sets, but tuples of other immutables can if you need to store compound values. Tuples compare by full values when used in sets:

```
>>> S = {1.23}

>>> S.add([1, 2, 3])              # Only immutable objects work in a set
TypeError: unhashable type: 'list'
>>> S.add({'a':1})
TypeError: unhashable type: 'dict'

>>> S.add((1, 2, 3))
>>> S                             # No list or dict; tuple, str, numbers OK
{1.23, (1, 2, 3)}

>>> S | {(4, 5, 6), (1, 2, 3)}    # Union: same as S.union(...)
{1.23, (4, 5, 6), (1, 2, 3)}

>>> (1, 2, 3) in S                # Membership: by complete values
True
>>> (1, 4, 3) in S
False
```

Tuples in a set, for instance, might be used to represent dates, records, IP addresses, and so on (more on tuples later in this part of the book). Sets may also contain modules, type objects, and more. Sets themselves are mutable too, and so cannot be nested in other sets directly; if you need to store a set inside another set, the `frozenset` built-in call works just like `set` but creates an immutable set that cannot change and thus can be embedded in other sets:

```
>>> S.add(frozenset('app'))
>>> S
{1.23, (1, 2, 3), frozenset({'a', 'p'})}
```

Set comprehensions

In addition to literals and calls, sets can also be made by running comprehension expressions, previewed briefly in [Chapter 4](#). Comprehensions also work for lists, dictionaries, and generators, and behave largely the same in all. For sets, comprehensions are coded in curly braces. When run, they perform a loop that collects the result of an expression on each iteration; a loop variable gives access to the current iteration value for use in the collection expression. The result is a new set with all the normal set behavior. For example:

```
>>> {x ** 2 for x in [1, 2, 3, 4]}           # Make a new set with a comprehension
{16, 1, 4, 9}
```

In this expression, the loop is coded on the right, and the collection expression is coded on the left ($x ** 2$). As for list comprehensions, we get back pretty much what this expression says: “Give me a new set containing X squared, for every X in a list.” Comprehensions can also iterate across other kinds of objects, such as strings; the first of the following examples also illustrates the comprehension-based way to make a set from an existing iterable:

```
>>> {x for x in 'py3X'}                     # Same as: set('py3X')
{'p', 'X', '3', 'y'}

>>> {c * 4 for c in 'py3X'}                 # Set of collected expression results
{'yyyy', '3333', 'XXXX', 'pppp'}

>>> {c * 4 for c in 'py3X' + 'py2X'}         # Expressions work on both sides
{'yyyy', '3333', 'XXXX', '2222', 'pppp'}

>>> S = {c * 4 for c in 'py3X'}             # All set ops work on results
>>> S | {'zzzz', 'XXXX'}
{'yyyy', '3333', 'XXXX', 'pppp', 'zzzz'}
>>> S & {'zzzz', 'XXXX'}
{'XXXX'}
```

Because the rest of the comprehensions story relies upon underlying concepts we’re not yet prepared to tackle, we’ll postpone further details until later in this book. In [Chapter 8](#), you’ll meet first cousins, the list and dictionary comprehension, and you’ll learn much more about all comprehensions—set, list, dictionary, and generator—later on, especially in [Chapters 14](#) and [20](#). As you’ll find, all comprehensions support additional syntax not shown here, including nested loops and `if` tests, which can be challenging before you’ve had a chance to study larger statements.

Why sets?

Set operations have a variety of common uses, some more practical than mathematical. For example, because items are stored only once in a set, sets can be used to *filter duplicates* out of other collections, albeit at the cost of original ordering because sets are unordered in general. Simply convert the collection to a set, and then convert it back:

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
>>> L = list(set(L))                         # Removing duplicates
>>> L
[1, 2, 3, 4, 5]

>>> list(set(['yy', 'cc', 'aa', 'xx', 'dd', 'aa'])) # But order may change
['xx', 'cc', 'yy', 'aa', 'dd']
```


Sets can be used to *isolate differences* in lists, strings, and other iterable objects too—simply convert to sets and take the difference—though again the unordered nature of sets means that the results may not match that of the originals:

```
>>> set([1, 3, 5, 7]) - set([1, 2, 4, 5, 6])      # Find list differences
{3, 7}
>>> set('abcdefg') - set('abdg hij')             # Find string differences
{'c', 'e', 'f'}
>>> set('code') - set(['t', 'o', 'e'])            # Find differences, mixed
{'c', 'd'}
```

You can also use sets to perform *order-neutral equality* tests by converting to a set before the test, because order doesn't matter in a set. More formally, two sets are *equal* if and only if every element of each set is contained in the other—that is, each is a subset of the other, regardless of order. For instance, you might use this to compare the outputs of programs that should work the same but may generate results in different order. Sorting with Python's `sorted` built-in before testing has the same effect for equality; sets don't rely on an expensive sort, but also don't order items:

```
>>> L1, L2 = [1, 3, 5, 2, 4], [2, 5, 3, 4, 1]
>>> L1 == L2                                     # Order matters in sequences
False
>>> set(L1) == set(L2)                           # Order-neutral equality
True
>>> sorted(L1) == sorted(L2)                     # Similar but results ordered
True
>>> 'code' == 'edoc', set('code') == set('edoc'), sorted('code') == sorted('edoc')
(False, True, True)
```

Sets can also be used to keep track of where you've already been when traversing a graph or other *cyclic* structure. For example, the transitive module reloader and inheritance-tree lister examples we'll code in Chapters 25 and 31, respectively, must keep track of items visited to avoid loops, as Chapter 19 discusses in the abstract. Using a list in this context is inefficient because searches require linear scans. Although recording states visited as keys in a dictionary is efficient, sets offer an alternative that's essentially equivalent and may be more intuitive.

Finally, sets are also convenient when you're dealing with large data collections like database query results—the intersection of two sets contains objects common to both categories, and the union contains all items in either set. To illustrate, here's a more tangible example of set operations at work, applied to people in a hypothetical company (like all examples in this book, any resemblance to the real world is purely coincidental!):

```
>>> engineers = {'pat', 'ann', 'bob', 'sue'}
>>> managers = {'sue', 'tom'}

>>> 'pat' in engineers                            # Is pat an engineer?
True

>>> engineers & managers                          # Who is both engineer and manager?
{'sue'}

>>> engineers | managers                         # All people in either category
{'ann', 'sue', 'pat', 'tom', 'bob'}

>>> engineers - managers                        # Engineers who are not managers
{'pat', 'ann', 'bob'}

>>> managers - engineers                       # Managers who are not engineers
{'tom'}
```

```

>>> engineers > managers          # Are all managers engineers? (superset)
False

>>> {'sue', 'bob'} < engineers     # Are both engineers? (subset)
True

>>> (managers | engineers) > managers  # All people is a superset of managers
True

>>> managers ^ engineers           # Who is in one group but not both?
{'ann', 'pat', 'tom', 'bob'}

```

You can find more details on set operations in the Python library manual and some mathematical and database texts. Also stay tuned for [Chapter 8](#)’s revival of set operations we’ve seen here, in the context of dictionary view objects. Here, we have time for just one last numeric object type.

Boolean Objects

Though somewhat gray, the Python Boolean type, `bool`, is arguably numeric in nature because its two values, `True` and `False`, are just customized versions of the integers 1 and 0 that print themselves differently. Python treats 1 and 0 as `true` and `false` like many programming languages, but its `True` and `False` makes Boolean roles more explicit. Although that’s all some programmers may need to know, let’s briefly reveal this type’s forgery.

To represent truth values, Python has an explicit Boolean type called `bool`, from which the objects pre-assigned to built-in names `True` and `False` are made. That is, `True` and `False` are instances of `bool`, which is in turn just a subclass (in the object-oriented sense) of the built-in integer type `int`. `True` and `False` behave exactly like the integers 1 and 0, except that they have customized printing logic—they print themselves as the words `True` and `False`, instead of the digits 1 and 0. `bool` accomplishes this by redefining `str` and `repr` string formats (introduced earlier in this chapter) for its two objects, and all logical tests yield `True` or `False` for their results.

Because of this customization, Boolean expressions typed at the interactive prompt print results as the words `True` and `False` instead of the less obvious 1 and 0. In addition, Booleans make truth values more apparent in your code. For instance, an infinite loop can be coded as `while True:` instead of the less intuitive `while 1:`, and flags can be initialized more clearly with `flag = False`. We’ll discuss these statements further in [Part III](#).

Again, though, for most practical purposes, you can treat `True` and `False` as though they are predefined variables set to integers 1 and 0. This implementation can lead to curious results, though; because `True` is just the integer 1 with a custom display format, `True + 4` yields integer 5 in Python:

```

>>> type(True)          # True is a bool
<class 'bool'>
>>> isinstance(True, int) # As well as an int
True
>>> True == 1           # Same value
True
>>> True is 1           # But a different object: see the next chapter!
False
>>> True or False       # Same as: 1 or 0
True

```

```
>>> True + 4          # (Hmmm)
5
```

Since you probably won't come across an expression like the last of these in real Python code, you can safely ignore any of its deeper metaphysical implications. We'll revisit Booleans in [Chapter 9](#) to define Python's notion of truth, and again in [Chapter 12](#) to see how Boolean operators like `and` and `or` work.

Numeric Extensions

Finally, although Python's core numeric objects offer plenty of power for most applications, a large catalog of third-party open source extensions is available to address more focused numeric needs.

We surveyed tools in this domain in [Chapter 1](#)'s section "What Can I Do with Python?" In short, there is a now-common stack of tools for advanced numeric coding in Python today, including *NumPy*, *SciPy*, *pandas*, *matplotlib*, *Jupyter*, and more, and additional tools address subdomains like statistics, astronomy, and AI.

This toolkit is used by research organizations, financial entities, and aerospace groups around the world, and performs the sort of tasks formerly coded in languages like C++ or Fortran. Many who work in this field liken the combination of Python plus numeric extensions to a free, flexible, and powerful alternative to systems like MATLAB.

Though a popular and exciting domain, Python numeric programming is just one way to use the language (Python web development, for example, is similarly sized) and is easily rich enough to fill entire books by itself. Hence, this book doesn't cover numeric extensions and focuses instead on teaching you the Python language that's used in every domain. Once you've learned Python itself, you'll find copious resources for add-ons both on the web and at book outlets near you when you're ready to level up.

Chapter Summary

This chapter has toured Python's numeric object types and the operations we can apply to them. Along the way, we met the trusty integer and floating-point objects, as well as some more exotic and less commonly used types such as complex numbers, decimals, fractions, and sets. We also explored Python's expression syntax, type conversions, bitwise operations, and various literal forms for coding numbers in scripts.

Later in this part of the book, we'll continue our in-depth object tour by filling in details about the next object type—the string. In the next chapter, however, we'll take some time to explore the mechanics of variable assignment in more detail than we have here. This turns out to be perhaps the most fundamental idea in Python, so make sure you check out the next chapter before moving on. First, though, it's time to take the usual chapter quiz.

Test Your Knowledge: Quiz

1. What is the value of the expression `2 * (3 + 4)` in Python, and why?
2. What is the value of the expression `2 * 3 + 4` in Python, and why?
3. What is the value of the expression `2 + 3 * 4` in Python, and why?
4. What tools can you use to find a number's square root, as well as its square?

5. What is the type of the result of the expression `1 + 2.0 + 3`, and why?
6. How can you truncate and round a floating-point number?
7. How can you convert an integer to a floating-point number?
8. How would you display an integer in octal, hexadecimal, or binary notation?
9. How might you convert an octal, hexadecimal, or binary string to a plain integer?

Test Your Knowledge: Answers

1. The value will be 14, the result of `2 * 7`, because the parentheses force the addition to happen before the multiplication.
2. The value will be 10, the result of `6 + 4`. Python's operator precedence rules are applied in the absence of parentheses, and multiplication has higher precedence than (i.e., happens before) addition, per [Table 5-2](#).
3. This expression yields 14, the result of `2 + 12`, for the same precedence reasons as in the prior question.
4. Functions for obtaining the square root, as well as *pi*, tangents, and more, are available in the imported `math` module. To find a number's square root, import `math` and call `math.sqrt(M)`. To get a number's square, use either the exponent expression `X ** 2` or the built-in function `pow(X, 2)`. Either of these last two can also compute the square root when given a power of 0.5 (e.g., `X ** .5`).
5. The result will be a floating-point number: the integers are converted up to floating point, the most complex type in the expression, and floating-point math is used to evaluate it.
6. The `int(N)` and `math.trunc(N)` functions truncate, and the `round(N, digits)` function rounds. We can also compute the floor with `math.floor(N)` and round for display with string-formatting operations.
7. The `float(I)` function converts an integer to a floating point; mixing an integer with a floating point within an expression will result in a conversion as well. In some sense, Python / true division converts too—it always returns a floating-point result that includes the remainder, even if both operands are integers.
8. The `oct(I)`, `hex(I)`, and `bin(I)` built-in functions return the octal, hexadecimal, and binary string forms for an integer. All three flavors of string formatting (expression, method, and f-string) also provide targets for some such conversions.
9. The `int(S, base)` function can be used to convert from octal, hexadecimal, and binary digit strings to normal integers (pass in 8, 16, or 2 for the *base*). The `eval(S)` function can be used for this purpose too, but it's more expensive to run and can have security risks. To some extent, other-base literals like `0xFFFF` and `0b1111` in your code do this work too when read by Python. Note that integers are always stored in binary form in computer memory; these are just display string format conversions.

The Dynamic Typing Interlude

In the prior chapter, we began exploring Python’s core object types in depth by studying Python numbers and their operations. We’ll resume our object type tour in the next chapter, but before we move on, it’s important that you get a handle on what may be the most fundamental idea in Python programming and is certainly the basis of much of both the conciseness and flexibility of the Python language: dynamic typing, and the polymorphism it implies.

As you’ll see here and throughout this book, in Python, we do not need to declare the specific types of the objects our scripts use. In fact, most programs should not care about specific types—*on purpose*. By avoiding constraints this way, code naturally works in many contexts and often more than expected. Because dynamic typing is the root of this flexibility, and is also a potential stumbling block for newcomers, let’s take a brief side trip to explore the model here. At the end of the trip, we’ll also make a short stop at the paradox of type hinting, to learn why you should avoid it.

The Case of the Missing Declaration Statements

If you have a background in statically typed languages like C, C++, or Java, you might find yourself a bit perplexed at this point in the book. So far, we’ve been using variables without declaring their existence or their types, and it somehow works. When we type `a = 3` in an interactive session or program file, for instance, how does Python know that `a` should stand for an integer? For that matter, how does Python know what `a` is at all?

Once you start asking such questions, you’ve crossed over into the domain of Python’s *dynamic typing* model. In Python, types are determined automatically at runtime (“dynamically”), not in response to declarations added to code ahead of time (“statically”). This means that you never need to declare variables—a concept that may be simpler to grasp if you keep in mind that it all boils down to variables, objects, and the links between them, as the next section explains.

Variables, Objects, and References

As you've seen in many of the examples used so far in this book, when you run an assignment statement such as `a = 3` in Python, it works even if you've never told Python to use the name `a` as a variable, or that `a` should stand for an integer-type object. In the Python language, this all pans out in a very natural way, as follows:

Variable creation

A variable (also known in Python as a name), like `a`, is created when your code first assigns it a value. Future assignments change the value of the already created name. Technically, Python detects some names before your code runs (e.g., locals in functions), but you can think of it as though initial assignments make variables.

Variable types

A variable itself never has any type information or constraints associated with it. In Python, the notion of type lives with objects, not names. Variables are generic in nature; they always simply refer to a particular object at a particular point in time.

Variable use

When a variable appears in an expression, it is immediately replaced with the object that it currently refers to, whatever that may be. Further, all variables must be explicitly assigned before they can be used; referencing unassigned variables results in errors.

In sum, variables are created when assigned, can reference any type of object, and must be assigned before they are referenced. This means that you never need to declare names used by your script, but you must initialize names before you can update them; counters, for example, must be initialized to zero before you can add to them.

This dynamic typing model is strikingly different from the typing model of traditional languages. When you are first starting out, the model is usually easier to understand if you keep clear the distinction between names and objects. For example, when we say this to assign a variable a value in a Python REPL or script:

```
>>> a = 3                # Assign a name to an object
```

at least conceptually, Python will perform three distinct steps to carry out the request. These steps reflect the operation of all assignments in the Python language:

1. Create an object to represent the value 3.
2. Create the variable `a`, if it does not yet exist.
3. Link the variable `a` to the new object 3.

The net result will be a structure inside Python that resembles [Figure 6-1](#). As sketched, variables and objects are stored in different parts of memory and are associated by links (the link is shown as a pointer in the figure). Variables always link to objects and never to other variables, but larger objects may link to other objects (for instance, a list object has links to the objects it contains).

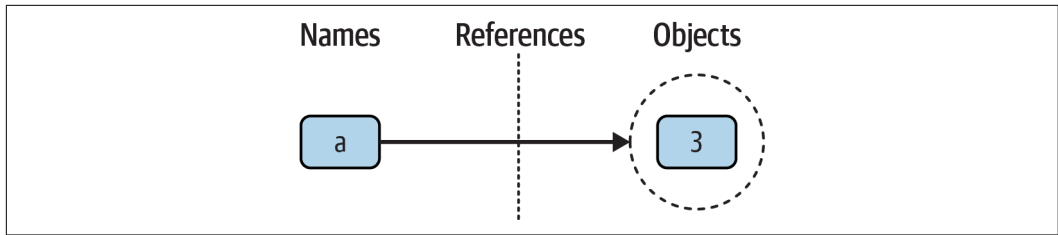


Figure 6-1. Names (a.k.a. variables) and objects after running the assignment `a = 3`

These links from variables to objects are called *references* in Python—a kind of association, implemented as an object’s address in memory. Whenever variables are later used (i.e., referenced), Python automatically follows the variable-to-object links. This is all simpler than the terminology may imply. In concrete terms:

- *Variables* are named entries in a system table, with spaces for links to objects.
- *Objects* are pieces of allocated memory, with enough space to represent the values for which they stand.
- *References* are automatically followed pointers from variables to objects.

At least conceptually, each time you generate a new value in your script by running an expression, Python creates a new *object* (i.e., a chunk of memory) to represent that value. As an optimization, Python internally caches and reuses certain kinds of unchangeable objects, such as small integers and strings (each 0 is not really a new piece of memory—more on this caching behavior later). But from a logical perspective, it works as though each expression’s result value is a distinct object and each object is a distinct piece of memory.

Technically speaking, objects have more structure than just enough space to represent their values. Each object also has two standard header fields: a *type designator* used to mark the type of the object, and a *reference counter* used to determine when it’s OK to reclaim the object. To understand how these two header fields factor into the model, we need to move on.

Python References for C Programmers

Readers with a background in C may find Python references similar to C *pointers* (i.e., memory addresses). In fact, references are implemented as pointers by CPython internally, and they often serve the same roles, especially with objects that can be changed in place (more on this later).

Because references are always automatically dereferenced when used, though, you can never actually *do* anything useful with a reference itself. As noted ahead, the referenced object’s address may be returned by the `id` built-in as a unique ID, but even this isn’t guaranteed: see Python’s manuals.

This lack of pointers avoids an entire category of notorious C bugs. But you can think of Python references as C “void*” pointers that are automatically followed when used, without going too far off base.

Types Live with Objects, Not Variables

To see how object types come into play, watch what happens if we assign a variable multiple times:

```
>>> a = 3                # It's an integer
>>> a = 'hack'           # Now it's a string
>>> a = 1.23             # Now it's a floating point
```

This isn't typical Python code, but it does work—a starts out as an integer, then becomes a string, and finally becomes a floating-point number. This example tends to look especially odd to ex-C programmers, as it appears as though the *type* of `a` changes from integer to string when we say `a = 'hack'`.

However, that's not really what's happening. In Python, things work more simply. *Names* have no types; as stated earlier, types live with objects, not names. In the preceding listing, we've simply changed `a` to reference different objects. Because variables have no type, we haven't actually changed the type of the variable `a`; we've simply made the variable reference a different type of object. In fact, again, all we can ever say about a variable in Python is that it references a particular object at a particular point in time.

Objects, on the other hand, know what type they are—each object contains a header field that tags the object with its type. The integer object 3, for example, will contain the value 3, plus a designator that tells Python that the object is an integer (strictly speaking, a pointer to an object called `int`, the name of the integer type). The type designator of the `'hack'` string object points to the string type (called `str`) instead, and 1.23 points to `float`. Because objects know their types, variables don't have to.

To recap, types are associated with objects in Python, not with variables. In typical code, a given variable usually will reference just one kind of object. Because this isn't a requirement, though, you'll find that Python code tends to be much more flexible than you may be accustomed to—if you use Python well, your code might work on many types automatically.

As mentioned, objects have two header fields, a type designator and a reference counter. To understand the latter of these, we need to move on and take a brief look at what happens at the end of an object's life.

Objects Are Garbage-Collected

In the prior section's listings, we assigned the variable `a` to different types of objects in each assignment. But when we reassign a variable, what happens to the value it was previously referencing? For example, after the following statements, what happens to the object 3?

```
>>> a = 3
>>> a = 'text'
```

The answer is that in Python, whenever a name is assigned to a new object, the space held by the prior object is reclaimed if it is not referenced by any other name or object. This automatic reclamation of objects' space is known as *garbage collection* and makes life much simpler for programmers of languages like Python that support it.

To illustrate, consider the following example, which sets the name `x` to a different object on each assignment:

```
>>> x = 99
>>> x = 'Python'          # Reclaim 99 now (unless referenced elsewhere)
>>> x = 3.1415            # Reclaim 'Python' now (ditto)
>>> x = [1, 2, 3]         # Reclaim 3.1415 now (ditto)
```


First, notice that `x` is set to a different type of object each time. Again, the effect is as though the type of `x` is changing over time, but this is not really the case. Remember, in Python types live with objects, not names. Because names are just generic references to objects, this sort of code works naturally.

Second, notice that references to objects are discarded along the way. Each time `x` is assigned to a new object, Python reclaims the prior object's space. For instance, when it is assigned the string `'Python'`, the object `99` is immediately reclaimed (assuming it is not referenced anywhere else)—that is, the object's space is automatically thrown back into the free space pool, to be reused for a future object.

Internally, Python accomplishes this feat by keeping a counter in every object that keeps track of the number of references currently pointing to that object. As soon as—and exactly when—this counter drops to zero, the object's memory space is automatically reclaimed. In the preceding listing, we're assuming that each time `x` is assigned to a new object, the prior object's reference counter drops to zero, causing it to be reclaimed.

The most immediately tangible benefit of garbage collection is that it means you can use objects liberally without ever needing to allocate or free up space in your script. Python will make objects clean up their unused space for you as your program runs. In practice, this eliminates a substantial amount of bookkeeping code required in lower-level languages such as C and C++.

More on Python Garbage Collection

Technically speaking, Python's garbage collection is based mainly upon reference counters, as described here, but it also has a component that detects and reclaims objects with *cyclic references* in time. This component can be disabled if you're sure that your code doesn't create cycles, but it is enabled by default. Both references and this optional component do garbage collection, but the latter may be what users of some other languages lacking reference counts think of as “garbage collection.”

Circular references are a classic issue in reference-count garbage collectors. Because references are implemented as pointers, it's possible for an object to reference itself, or reference another object that does. For example, exercise 6 in “[Test Your Knowledge: Part I Exercises](#)” on page 48 and its solution in [Appendix B](#) show how to create a cycle easily by embedding a reference to a list within itself (e.g., `L.append(L)`). The same phenomenon can occur for assignments to attributes of objects created from user-defined classes. Though relatively rare, because the reference counts for such objects never drop to zero, they must be treated specially.

For more details on Python's cycle detector and collector, see the documentation for the `gc` module in Python's library manual. The best takeaway here is that garbage-collection-based memory management is implemented for you in Python, by people highly skilled at the task; it works well, even for cycles.

Also note that this chapter's description of Python's garbage collector is not part of the language's definition and applies to the standard implementation of Python (a.k.a. *CPython*) only. [Chapter 2](#)'s alternative implementations such as *Jython*, *IronPython*, and *PyPy* may use different schemes, though the net effect in all is similar—unused space is reclaimed for you automatically, if not always as immediately.

Shared References

So far, we've explored what happens as a single variable is assigned references to objects. Now let's introduce another variable into our interaction and watch what happens to its names and objects:

```
>>> a = 3
>>> b = a
```

Typing these two statements generates the scene captured in [Figure 6-2](#). The second command causes Python to create the variable `b`; the variable `a` is being used and not assigned here, so it is replaced with the object it references (3), and `b` is made to reference that object. The net effect is that the variables `a` and `b` wind up referencing the *same object*—that is, pointing to the same chunk of memory.

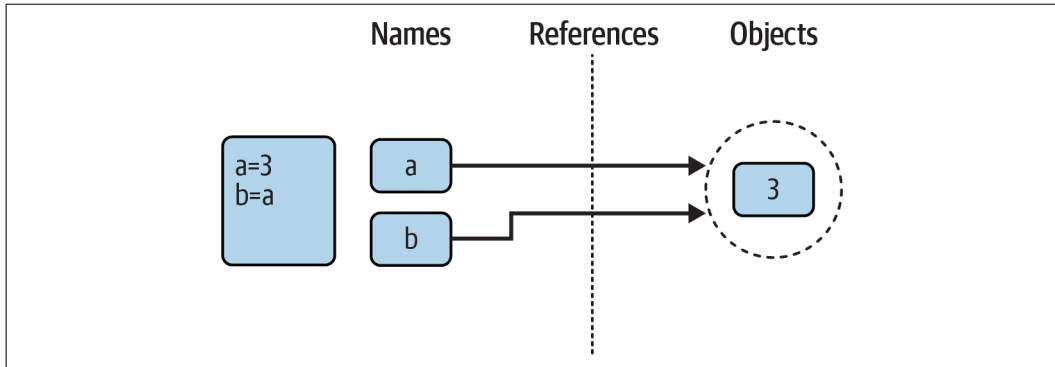


Figure 6-2. Names and objects after next running the assignment `b = a`

This scenario in Python—with multiple names referencing the same object—is usually called a *shared reference* (and sometimes and perhaps more accurately, *shared object*). Note that the names `a` and `b` are not linked to each other directly when this happens; in fact, there is no way to ever link a variable to another variable in Python. Rather, both variables point to the same object via their references.

Next, suppose we extend the session with one more statement:

```
>>> a = 3
>>> b = a
>>> a = 'hack'
```

As with all Python assignments, this statement simply makes a new object to represent the string value `'hack'` and sets `a` to reference this new object. It does not, however, change the value of `b`; `b` still references the original object, the integer 3. The resulting reference structure is shown in [Figure 6-3](#).

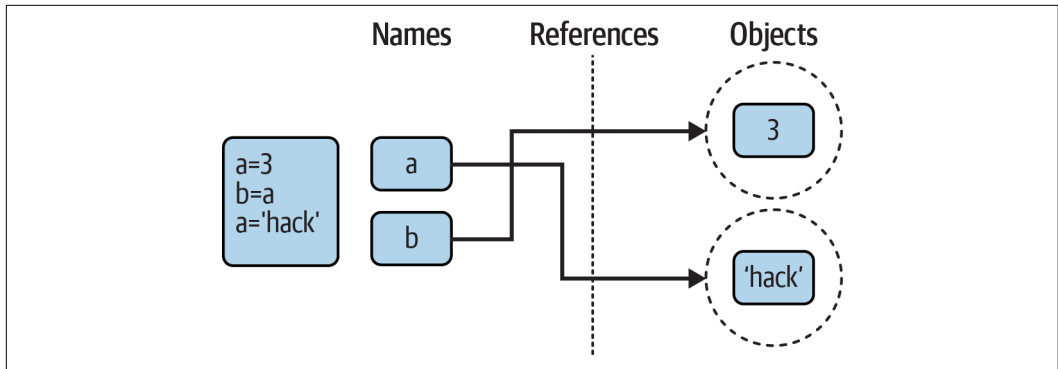


Figure 6-3. Names and objects after finally running the assignment `a = 'hack'`

The same sort of thing would happen if we changed `b` to `'hack'` instead—the assignment would change only `b`, not `a`. This behavior also occurs if there are no type differences at all. For example, consider these three statements:

```
>>> a = 3
>>> b = a
>>> a = a + 2
```

In this sequence, the same events transpire. Python makes the variable `a` reference the object `3` and makes `b` reference the same object as `a`, as in Figure 6-2; as before, the last assignment then sets `a` to a completely different object (in this case, the integer `5`, which is the result of the `+` expression). It does not change `b` as a side effect. In fact, there is no way to *ever* overwrite the value of the object `3`—as introduced in Chapter 4, integers are *immutable* and thus can never be changed in place (see why this stuff matters?). All we can ever do is make a new integer object.

One way to think of this is that, unlike in some languages, variables in Python are always pointers to objects, not labels of changeable memory areas: setting a variable to a new value does not alter the original object, but rather causes the variable to reference an entirely different object. The net effect is that assignment to a variable itself can impact only the single variable being assigned. When mutable objects and in-place changes enter the equation, though, the picture changes somewhat; to see how, let's move on.

Shared References and In-Place Changes

As you'll learn more in this part's upcoming chapters, some operations do change objects *in place*, but they're only supported by Python's *mutable* types—including lists, dictionaries, and sets. For instance, an assignment to an offset in a list actually changes the list object itself in place, rather than generating a brand-new list object.

Though you must take it somewhat on faith at this point in the book, this distinction can matter much in your programs. For objects that support such in-place changes, you need to be more aware of shared references, since a change from one name may impact others. Otherwise, your objects may seem to change for no apparent reason. Given that all assignments are based on references (including function argument passing), it's a pervasive phenomenon.

To illustrate, let's take another look at the list objects introduced in [Chapter 4](#). Recall that lists, which do support in-place assignments to positions, are simply collections of other objects, coded in square brackets:

```
>>> L1 = [2, 3, 4]
>>> L2 = L1
```

L1 here is a list containing the objects 2, 3, and 4. Items inside a list are accessed by their positional offsets, so L1[0] refers to object 2, the first item in the list L1. Of course, lists are also objects in their own right, just like integers and strings. After running the two prior assignments, L1 and L2 reference the same shared object, just like a and b in the prior example (see [Figure 6-2](#)). Now imagine that, as before, we extend this interaction to say the following:

```
>>> L1 = 24
```

This assignment simply sets L1 to a different object; L2 still references the original list much as in the preceding section. If we change this statement's syntax slightly, however, it changes its effect radically:

```
>>> L1 = [2, 3, 4]          # A mutable object
>>> L2 = L1                # Make a reference to the same object
>>> L1[0] = 24              # An in-place change

>>> L1                      # L1 is different
[24, 3, 4]
>>> L2                      # But so is L2!
[24, 3, 4]
```

Really, we haven't changed L1 itself at line three here; we've changed a component of the *object* that L1 references. This sort of change overwrites part of the list object's value in place. Because the list object is shared by (referenced from) other variables, though, an in-place change like this doesn't affect only L1—that is, you must be aware that when you make such changes, they can impact other parts of your program. In this example, the effect shows up in L2 as well because it references the same object as L1. Again, we haven't actually changed L2, either, but its value will appear different because it refers to an object that has been overwritten in place.

This behavior occurs only for mutable objects that support in-place changes and is usually what you want, but you should be aware of how it works so that it's expected. It's also just the default: if you don't want such behavior, you can request that Python *copy* objects instead of making references. There are a variety of ways to copy a list, including using the built-in `list` function, the `list` copy method, and the standard-library `copy` module. Perhaps the most common way is to slice from start to finish (see [Chapters 4](#) and [7](#) for more on slicing):

```
>>> L1 = [2, 3, 4]
>>> L2 = L1[:]              # Make a copy of L1 (or list(L1), L1.copy(), etc.)
>>> L1[0] = 24

>>> L1
[24, 3, 4]
>>> L2                      # L2 is not changed this time: different objects
[2, 3, 4]
```

Here, the change made through L1 is not reflected in L2 because L2 references a copy of the object L1 references, not the original; that is, the two variables point to different objects and different pieces of memory.

Note that this slicing technique won't work on the other major mutable core types, dictionaries and sets, because they are not sequences—to copy a dictionary or set, instead use their `X.copy()` method call

(lists have one too), or pass the original object to their type names, `dict` and `set`. Also, note that the standard-library `copy` module has a call for copying any object type generically, as well as a call for copying nested object structures—a dictionary with nested lists, for example:

```
import copy
X = copy.copy(Y)           # Make top-level "shallow" copy of any object Y
X = copy.deepcopy(Y)       # Make deep copy of any object Y: copy all nested parts
```

We'll explore lists and dictionaries in more depth, and revisit the concept of shared references and copies, in Chapters 8 and 9. For now, keep in mind that objects that can be changed in place—that is, mutable objects—are always open to these kinds of effects in any code they pass through. In Python, this includes lists, dictionaries, sets, and some objects defined with `class` statements. If this is not desired behavior, simply copy your objects as needed.

Shared References and Equality

In the interest of full disclosure, it's worth pointing out that the garbage-collection behavior described earlier in this chapter may be more conceptual than literal for certain types. Consider these statements:

```
>>> x = 99
>>> x = 'Python'           # Reclaim 99 now?
```

Because Python caches and reuses small integers and small strings, as mentioned earlier, the object 99 here is probably not literally reclaimed; instead, it will likely remain in a system table to be reused the next time you generate a 99 in your code. Most kinds of objects, though, are reclaimed immediately when they are no longer referenced; for those that are not, the caching mechanism is irrelevant to your code—unless you use atypical tools.

For instance, because of Python's reference model, there are two different ways to check for *equality* in a Python program. Let's create a shared reference to demonstrate:

```
>>> L = [1, 2, 3]
>>> M = L                   # Make M and L reference the same object
>>> L == M                  # Same values
True
>>> L is M                  # Same objects
True
```

The first technique here, the `==` operator, tests whether the two referenced objects have the same *values*; this is the method almost always used for equality checks in Python. The second method, the `is` operator, instead tests for object *identity*—it returns `True` only if both names point to the exact same object, so it is a much stronger form of equality testing and is rarely applied in most programs (except for single-instance objects like `None`, `True`, and `False`, as in the prior chapter).

Really, `is` simply compares the pointers that implement references, and it serves as a way to detect shared references in your code if needed. It returns `False` if the names point to equivalent but different objects, as is the case when we run two different literal expressions:

```
>>> L = [1, 2, 3]
>>> M = [1, 2, 3]           # Make M and L reference different objects
>>> L == M                  # Same values
True
>>> L is M                  # Different objects
False
```

But now watch what happens when we perform the same operations on an immutable object like an integer:

```
>>> X = 99
>>> Y = 99                # Should be two different objects
>>> X == Y
True
>>> X is Y                # Same object anyhow: caching at work!
True
```

In this interaction, `X` and `Y` should be `==` (same value), but not `is` (same object) because we ran two different literal expressions (`99`). Because some integers and strings are cached and reused, though, `is` tells us they reference the same single object.

If you really want to look under the hood, the `id` built-in is another way to check object identities (and may or may not return an object's address in memory); and the `getrefcount` function in the standard `sys` module returns the passed object's reference count. As of Python 3.12, however, the latter is not as interesting as it once was, because it returns a very high count for objects considered to be immortal—which in Python just means cached for reuse:

```
>>> import sys
>>> sys.getrefcount(99)      # 99 is immortal (cached)
4294967295
>>> sys.getrefcount(2 ** 1000) # But this is not
1
>>> id(99) == id(99)        # Same ID/same object (address?)
True
```

This object caching and reuse is irrelevant to your code (unless you run the `is` check!). Because you cannot change *immutable* numbers or strings in place, it doesn't matter how many references there are to the same object—every reference will always see the same, unchanging value even if they all reference the same cached object. Still, this behavior reflects one of the many ways Python optimizes its model for execution speed.

Dynamic Typing Is Everywhere

Of course, you don't really need to draw name/object diagrams with circles and arrows to use Python. When you're starting out, though, it sometimes helps you understand unusual cases if you can trace their reference structures as we've done here. If a mutable object changes out from under you when passed around your program, for example, chances are you are witnessing some of this chapter's subject matter firsthand.

Moreover, even if dynamic typing seems a little abstract at this point, you probably will care about it eventually. Because *everything* seems to work by assignment and references in Python, a basic understanding of this model is useful in many different contexts. As you'll see, it works the same in assignment statements, function arguments, for loop variables, module imports, class attributes, and more. The good news is that there is just *one* assignment model in Python; once you get a handle on dynamic typing, you'll find that it works the same everywhere in the language.

At the most practical level, dynamic typing means there is less code for you to write. Just as importantly, though, dynamic typing is also the root of Python's *polymorphism*, a concept introduced in [Chapter 4](#) that we'll revisit again later in this book. Because we do not constrain types in Python code, it is both concise and highly flexible. As you'll see, when used well, dynamic typing—and the polymorphism it implies—produces code that automatically adapts to new requirements as your systems evolve.

Type Hinting: Optional, Unused, and Why?

Finally, an implausible plot twist. If you've read Python code written in recent years, you may have stumbled across some type declarations for variable names that look like the following—and seem curious and out of place for a dynamically typed language like Python, and at first glance contradictory to some of this chapter's claims:

```
>>> a: int
>>> b: int = 0
>>> c: list[int] = [1, 2, 3]
```

As previewed in [Chapter 4](#), this is known as *type hinting*. Syntactically, it takes the form of a colon and object type, between a variable and an optional assignment. The object type can be a name or an expression to denote collections (`list[int]` means a list of integers) and can use names predefined in the standard-library typing module (e.g., `Iterable`, `Union`, and `Any`) to express richer types per elaborate theory. As of Python 3.12, a new type statement can even define type aliases to use in hints, though simple assignments that predated it can too:

```
>>> type Data = list[float]
>>> Data = list[float]
```

As also noted in [Chapter 4](#), though, type hints are optional, unused, and largely *academic*. Python does not require them and does not use them in any way and has no intentions of ever doing so. They are meant solely for use in third-party *tools* like type checkers, and as a form of *documentation* that's an alternative to code comments. You can say the same things more simply in both `#` comments and documentation strings you'll meet later.

Even when used, type hints do not constrain your code's types in any way. The preceding type hint for `a`, for instance, does not create name `a` (only assignment does), and `b`'s and `c`'s hints are not enforced in the least:

```
>>> a
NameError: name 'a' is not defined
>>> b = 'hack'
>>> c = 'code'
>>> b, c
('hack', 'code')
```

Type hints can also appear in definitions of functions (and class methods) to document types of parameters and results, commandeering an earlier feature known as function *annotations*. We haven't covered these yet, but as a preview, the following function hints that it accepts an integer and list of strings and returns a float—extraneous info that shows up in `__annotations__` dictionaries of hosting objects:

```
>>> def func(a: int, b: list[str]) -> float:
    return 'anything' + a + b
```

Yet as for simple variables, these hints are fully unused, and anything goes when this function is actually run. Strings, for example, work fine for both inputs and outputs, despite the seemingly rigid hints:

```
>>> func('You', 'Want')
'anythingYouWant'
```

That is, type hinting is a conceptually heavy tool adopted by Python but *completely unused by Python*. It's at best just another form of documentation in Python itself, albeit one that comes with complex rules. External tools might use type hints to check for type mismatches (e.g., *mypy*) or boost performance, but such tools are also optional, uncommon, and wholly separate from the Python language.

Furthermore, programs require runtime testing in any language, and optimized Pythons introduced in [Chapter 2](#) do not use type hints today, and in some cases cannot (see *PyPy*).

More to the point, though, type hinting is also *completely at odds* with Python's core notion of dynamic typing. Type declarations in a dynamically typed language are a pointless paradox that negates much of Python's value proposition. Teaching this bizarre extension to Python learners would be a disservice to both Python and learners.

Hence, this book recommends that beginners avoid type hinting at least until they are comfortable with Python's dynamic-typing paradigm. This book also won't be covering it further, because it's far too much extra heft sans benefit for newcomers struggling to master Python's already sizable fundamentals. If and when you opt to delve into this inane yet convoluted corner of Python, consult its docs for more information.

In the end—and despite what you may see in Python code written by programmers coming from other languages—type hinting does not mean that Python is statically typed. Python still uses only dynamic typing, and hopefully always will. After all, this is the root of most of its advantages over other tools. Let's hope that Python developers of the future learn this well before bloating or breaking a tool used and beloved by millions.

Chapter Summary

This chapter took a deeper look at Python's dynamic typing model—that is, the way that Python keeps track of object types for us automatically, rather than requiring us to code declaration statements in our scripts.

Along the way, we learned how variables and objects are associated by references in Python that enable type flexibility. We also explored the topic of garbage collection, learned how shared references to mutable objects can affect multiple variables, and saw how references impact the notion of equality in Python. Lastly, we briefly glimpsed type hinting—a subdomain that weirdly adds unused type declarations to a dynamically typed language.

Because there is just one assignment model in Python, and because assignment pops up everywhere in the language, it's important that you have a handle on the model before moving on. The following quiz should help you review some of this chapter's ideas. After that, we'll resume our core object tour in the next chapter, with strings.

Test Your Knowledge: Quiz

1. Consider the following three statements. Do they change the value printed for A?

```
A = 'code'
B = A
B = 'Python'
```

2. Consider these three statements. Do they change the printed value of A?

```
A = ['code']
B = A
B[0] = 'Python'
```


3. How about these—is A changed now?

```
A = ['code']  
B = A[:]  
B[0] = 'Python'
```

Test Your Knowledge: Answers

1. No: A still prints as `'code'`. When B is assigned to the string `'Python'`, all that happens is that the variable B is reset to point to the new string object. A and B initially share (i.e., reference/point to) the same single string object `'code'`, but two names are never linked together in Python. Thus, setting B to a different object has no effect on A. The same would be true if the last statement here were `B = B + 'coding'`, by the way—the concatenation would make a new object for its result, which would then be assigned to B only. We can never overwrite a string (or number, or tuple) in place, because strings are immutable.
2. Yes: A now prints as `['Python']`. Technically, we haven't really changed either A or B; instead, we've changed part of the object they both reference (point to) by overwriting that object in place through the variable B. Because A references the same object as B, the update is reflected in A, too.
3. No: A still prints as `['code']`. The in-place assignment through B has no effect this time because the slice expression made a copy of the list object before it was assigned to B. After the second assignment statement, there are two different list objects that have the same value—in Python, we say they are `==`, but not `is`. The third statement changes the value of the list object pointed to by B, but not that pointed to by A.

When References Are “Weak”

You may occasionally see the term “weak reference” in the Python world. No, this term isn't a judgment about inferiority. It refers to a somewhat obscure and advanced tool, which is related to the reference model we've explored here, and like the `is` operator, can't really be understood without it.

In short, a *weak reference*, implemented by the `weakref` standard-library module, is a reference to an object that does not by itself prevent the referenced object from being garbage-collected. If the last remaining references to an object are all weak references, the object can be reclaimed. When this happens, the weak references to it will be notified that the object no longer exists and can respond as needed.

As an example of its utility, this can be useful in nonessential *caches* of large objects primarily used elsewhere. If such a cache uses normal references, the cache's references alone would keep the objects in memory indefinitely. By using weak references, the object's space may be reclaimed when it's no longer needed for its primary role, and the cache will be notified of its demise, either on next fetch or by callback.

Not all object types can be weakly referenced, though support can be added for some with OOP techniques we won't explore till later in this book. Still, this is really just a special-case extension to the reference model we met here. For more details on weak references, see Python's library-manual coverage of `weakref`, a useful—if unhappily named—tool.

String Fundamentals

So far, we’ve studied numbers and explored Python’s dynamic typing model. The next major type on our in-depth object tour is the Python *string*—an ordered collection of characters used to store and represent text- and bytes-based information. We looked briefly at strings in [Chapter 4](#). Here, we will revisit them to fill in details we skipped earlier.

Before we get started, let’s get clear on what we *won’t* be covering here. [Chapter 4](#) also briefly previewed *Unicode* strings and files—tools for dealing with non-ASCII text. Unicode is a key tool for programmers, especially those who work in the internet domain. It can pop up, for example, in web pages, emails, GUI toolkits, file-processing tools, XML and JSON text, and more. At the same time, Unicode can be a heavy topic for programmers just starting out, and a complete understanding of it relies on tools that we haven’t yet studied in full, like files.

In light of that, this book splits its strings coverage between the essentials here, and their extension to Unicode and byte strings in [Chapter 37](#) of its advanced topics part. That is, this chapter tells only part of the string story in Python—the part that most scripts use, and most Python learners need to know up front. Despite this limited scope, everything we learn here will apply directly to Unicode and bytes processing, too, because Python text strings *are* Unicode, even if they’re simple ASCII text, and byte strings are simply strings constrained to byte values.

After you’ve learned the basics here, [Chapter 37](#) is recommended reading for the rest of the string saga, and most programmers will want to follow up with its coverage eventually. Unicode is rarely optional in programming today, though it’s best deferred until you’ve had a chance to master strings in general. So let’s get started!

String Object Basics

From a functional perspective, strings can be used to represent just about anything that can be encoded as text or bytes. In the text department, this includes symbols and words (e.g., your name), contents of text files loaded into memory, internet addresses, Python source code, and so on. Strings can also be used to hold the raw bytes used for media files and network transfers, and both the encoded and decoded forms of non-ASCII Unicode text.

You may have used strings in other languages, too. Python’s strings serve the same role as character arrays in languages such as C, but they are a somewhat higher-level tool than arrays. Unlike in C, strings in Python come with a powerful set of precoded processing tools. Also unlike languages such as

C, Python has no distinct type for individual characters; instead, you just use one-character strings for one-character info.

Strictly speaking, Python strings are categorized as *immutable sequences*, meaning that the characters they contain have a left-to-right positional order and cannot be changed in place. In fact, strings are the first representative of the larger class of objects called *sequences* that we will explore here. Pay attention to the sequence operations covered in this chapter, because they will work the same on other sequence types you'll meet later, such as lists and tuples.

As a first step, [Table 7-1](#) previews common string literals and operations discussed in this chapter, by abstract example (don't expect its code snippets to run!). As it shows, empty strings are written as a pair of quotation marks (single or double) with nothing in between, and there are a variety of ways to code strings. For processing, strings support *expression* operations such as concatenation (combining strings), slicing (extracting sections), indexing (fetching by offset), and so on. Besides expressions, Python also provides a set of string *methods* that implement common string-specific tasks, as well as *modules* for more advanced text-processing tasks such as pattern matching.

Table 7-1. Common string literals and operations

Operation	Interpretation
<code>S = ''</code>	Empty string
<code>S = "app's"</code>	Double quotes, same as single
<code>S = 'c\no\td\x00e'</code>	Escape sequences
<code>S = """...multiline..."""</code>	Triple-quoted block strings
<code>S = r'\temp\data.txt'</code>	Raw strings (ignore escapes)
<code>B = b'h\xc4ck'</code>	Byte strings (Chapter 4 , Chapter 37)
<code>S = 'py\u0001F40D'</code>	Unicode strings (Chapter 4 , Chapter 37)
<code>S = u'py\u0001F40D'</code>	Python 2.X compatibility (Chapter 37)
<code>S1 + S2</code> <code>S * 3</code>	Concatenate, repeat
<code>S[i]</code> <code>S[i:j]</code> <code>len(S)</code>	Index, slice, length
<code>S1 > S2</code> , <code>S1 == S2</code>	Comparisons: magnitude, equality
<code>'a %s coder' % kind</code>	String-formatting expression
<code>'a {0} coder'.format(kind)</code>	String-formatting method
<code>f'a {kind} coder'</code>	String-formatting literal (3.6+)
<code>S.find('od')</code> <code>S.rstrip()</code> <code>S.replace('od', 'ood')</code> <code>S.split(',')</code> <code>S.isdigit()</code> <code>S.lower()</code> <code>S.endswith('thon')</code> <code>S.join(strlist)</code> <code>S.encode('utf8')</code> <code>B.decode('latin1')</code>	String methods (see ahead for all 43): search, remove whitespace, replacement, split on delimiter, content test, case conversion, end test, delimiter join, Unicode encoding, Unicode decoding, etc. (see Table 7-3)

Operation	Interpretation
'py' in S.lower() for x in S: print(x) [c * 2 for c in S] map(ord, S)	Membership, iteration
re.match('Py(.*?)on', line)	Pattern matching: library module

Beyond the core set of string tools in [Table 7-1](#), Python also supports more advanced pattern-based string processing with the standard library's `re` (for “regular expression”) module demoed in [Chapter 37](#), and even higher-level text processing tools such as HTML, JSON, and XML parsers. This book and this chapter, though, are focused on the fundamentals represented by [Table 7-1](#).

This chapter begins with an overview of string literal forms and string expressions, then moves on to look at more advanced tools such as string methods and formatting. Python comes with many string tools, and we won't cover them all here; the complete story is chronicled in the Python library manual. Our goal here is to explore enough commonly used tools to give you a representative sample; methods we won't see in action here are analogous to those we will.

String Literals

By and large, strings are fairly easy to use in Python. The first thing you need to know about them, though, is that there are many ways to write them in your code:

- Single quotes: `'cod"e'`
- Double quotes: `"cod'e"`
- Triple quotes: `'''...code...'''`, `"""...code..."""`
- Escape sequences: `"c\tto\nd\0e"`
- Raw strings: `r"C:\new\test.bin"`
- ([Chapter 37](#)) Bytes literals: `b'co\x01de'`
- ([Chapter 37](#)) Unicode literals: `'h\u00c4ck'`

The single- and double-quoted forms are by far the most common; the others serve specialized roles, and we're postponing further discussion of the last two advanced forms until [Chapter 37](#). Let's take a quick look at all the other options in turn.

Single and Double Quotes Are the Same

Around Python strings, single- and double-quote characters are interchangeable, though they must match, and must be straight quotes (beware tools that autocorrect to slanted quotes!). That is, string literals can be written enclosed in either two single or two double straight quotes—the two forms work the same and return the same type of object. For example, the following two strings, coded at the usual REPL, are identical once they are read by Python:

```
$ python3
>>> 'python', "python"
('python', 'python')
```

The reason for supporting both is that it allows you to embed a quote character of the other variety inside a string without escaping it with a backslash. You may embed a double-quote character in a

string enclosed in single-quote characters, and vice versa, without having to use the escapes you'll meet in a moment:

```
>>> 'python"s', "python's"           # Mixed quotes sans escapes
('python"s', "python's")
```

This book generally prefers to use *single* quotes around strings just because they are marginally easier to read, except in cases where a single quote is embedded in the string. This is a purely subjective style choice, but Python displays strings this way, too, and most Python programmers do the same today, so you probably should too.

Note that the comma is important in the preceding code. Without it, Python *automatically concatenates* adjacent string literals of any kind. It is almost as simple to add a + operator between them to invoke concatenation explicitly, but adjacent literals are concatenated early when your code is read (and as you'll learn ahead, wrapping this form in parentheses also allows it to span multiple lines for larger blocks of text that can't use triple quotes):

```
>>> title = "Learning " 'Python' " 6E"   # Implicit concatenation when read
>>> title
'Learning Python 6E'
```

Adding commas between these strings would result in a tuple, not a string. Also notice in all of these outputs that Python prints strings in single quotes unless they embed one. If needed, you can also embed quote characters by escaping them with backslashes:

```
>>> 'python\'s', "python\'s"
('python's', 'python's')
```

To understand why, you need to move on to learn how escapes work in general.

Escape Sequences Are Special Characters

The prior example embedded a quote inside a string by preceding it with a backslash (\). This is representative of a general pattern in strings: backslashes are used to introduce special character codings known as *escape sequences*.

Escape sequences let us embed characters in strings that cannot easily be inserted into a string literal or typed on a keyboard. The character \, and one or more characters following it in the string literal, are replaced with a *single* character in the resulting string object, which has the value specified by the escape sequence. For example, here is a five-character string that embeds a newline and a tab:

```
>>> s = 'a\nb\tc'
```

The two characters \n stand for a single character—the *newline* character (technically speaking, code-point value 10, which means newline in Unicode and its ASCII subset). Similarly, the sequence \t is replaced with the *tab* character (code point 9). The way this string looks when printed depends on how you print it. The interactive echo shows the special characters as escapes, but `print` interprets them instead:

```
>>> s
'a\nb\tc'
>>> print(s)
a
b      c
```

To be completely sure how many actual characters are in this string, use the built-in `len` function—it returns the actual number of characters in a string, regardless of how it is coded or displayed:

```
>>> len(s)
5
```

This string is five characters long: it contains an ASCII `a`, a newline character, an ASCII `b`, and so on.



But length is not bytes: If you’re accustomed to all-ASCII text, it’s tempting to think of this `len` result as meaning five *bytes* too, but you probably shouldn’t. Really, “bytes” in today’s *Unicode* world doesn’t hold the meaning it once did. For one thing, the Python string object includes admin data that makes it larger in memory than its text alone.

For another, string content and length both reflect *code points* (identifying numbers) assigned by Unicode, and a single character’s code point does not necessarily map to a single byte—either when decoded in memory or encoded in files. Under encoding UTF-16, for example, ASCII characters are multiple bytes in files and may be any size at all in memory depending on how Python allocates their space. Moreover, code-point values of non-ASCII characters like 🐍 and 👍 are far too large to fit in an 8-bit byte in any form.

In fact, Python defines `str` strings formally as *sequences of Unicode code points*, not bytes, to make this clear. There’s much more on how Unicode text obviates bytes in [Chapter 37](#) when you’re ready to take the plunge. For now, to avoid confusion, think *characters* instead of *bytes* in strings.

Notice that the original backslash characters in the preceding examples are not really stored with the string in memory; they are used only to describe special character values to be stored in the string. For coding such special characters, Python recognizes a full set of escape code sequences, listed for reference in [Table 7-2](#).

Table 7-2. String backslash characters

Escape	Meaning
<code>\\</code>	Backslash (stores one <code>\</code>)
<code>\'</code>	Single quote (stores <code>'</code>)
<code>\"</code>	Double quote (stores <code>"</code>)
<code>\n</code>	Newline (a.k.a. linefeed)
<code>\r</code>	Carriage return (e.g., in Windows <code>\r\n</code>)
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\a</code>	Bell (where supported)
<code>\b</code>	Backspace
<code>\f</code>	Formfeed
<code>\xhh</code>	Hexadecimal code-point or byte value (exactly 2 hex digits)
<code>\ooo</code>	Octal code-point or byte value (up to 3 digits, 377 ceiling)

Escape	Meaning
<code>\0</code>	Null: octal binary 0 character (doesn't end string)
<code>\uhhhh</code>	Unicode character code point, 16-bit value (exactly 4 hex digits)
<code>\Uhhhhhhhh</code>	Unicode character code point, 32-bit value (exactly 8 hex digits)
<code>\N{name}</code>	Character with ID <i>name</i> in Unicode database
<code>\newline</code>	Ignored (precedes a continuation line)
<code>\other</code>	Retained verbatim, but a warning in 3.12, an error in the future

Some of [Table 7-2](#)'s escapes come with usage rules. Again for reference, here are the fine points:

- The `\x`, `\u`, and `\U` escape sequences require exactly two, four, and eight hexadecimal digits, respectively. Use digits 0–9 and A–F (uppercase or lowercase) for *h*.
- The `\o` escape accepts one to three octal digits and issues a warning for values over `\377` in Python 3.12 because values too big for a byte cause issues in byte strings. Use digits 0–7 for *o*.
- Both `\u` and `\U` are recognized only in `str` text-string literals (e.g., `'...'`), where they give a character's Unicode code-point value. This is the code point's decoded value.
- `\x` and `\o` escapes work in both `bytes` byte-string literals (`b'...'`), where they give a byte's absolute value; and in `str` text-string literals, where they give a character's Unicode code-point value.
- Lettered escapes like `\n` stand for their Unicode code points in text and their ASCII encodings in bytes, even if the two values agree (there is more on Unicode escapes in [Chapter 37](#)).

Let's get back to running code. Some string escape sequences allow you to embed absolute values as characters of a string. When you do this, you're really giving the *code-point* value of the desired character. For instance, here's a five-character string that embeds two characters with zero values (coded as octal escapes of one digit):

```
>>> s = 'a\0b\0c'
>>> s
'a\x00b\x00c'
>>> len(s)
5
```

The character associated with code point 0 is generally known as NULL (or NUL). Importantly, in Python, a character like this does not terminate a string the way a “null byte” typically does in C. Instead, Python keeps both the string's length and text in memory. In fact, *no* character terminates a string in Python. Here's a string that is all absolute escape codes—an absolute 1 and 2 (coded in octal), followed by an absolute 3 (coded in hexadecimal), and nonprintables all:

```
>>> s = '\001\002\x03'
>>> s
'\x01\x02\x03'
>>> len(s)
3
```


Notice that Python displays nonprintable characters in hex, regardless of how they were specified. When needed, you can freely combine characters, absolute-value escapes, and the more symbolic escapes in [Table 7-2](#). To demo, the following string contains the characters “HACK”, a tab and newline, and an absolute zero character coded in hex:

```
>>> S = 'H\tA\nC\x00K'
>>> S
'H\tA\nC\x00K'
>>> len(S)
7
>>> print(S)
H   A
CK
```

This becomes more important to know when you process binary data files in Python. Because their contents are represented as strings in your scripts, it's OK to process binary files that contain any sorts of binary byte values. When opened in binary modes, files return raw bytes from the external file as bytes—a string variant that supports most of the syntax and tools in this chapter (you'll find more on files and bytes in [Chapters 4, 9, and 37](#)).

Two limits in [Table 7-2](#) merit callouts. First of all, *octal* escapes with values too large for a byte issue warnings as of Python 3.12 and will be errors soon—despite the fact that these escapes in *text* strings denote code points, not bytes:

```
>>> '\400'
<stdin>:1: SyntaxWarning: invalid octal escape sequence '\400'
'Ä'
```

This seems unlikely to break much code, but the last entry in [Table 7-2](#) may: if Python does not recognize the character after a `\` as being a valid escape code, it simply keeps the backslash in the resulting string—at least for the moment:

```
>>> x = 'C:\py\code'
<stdin>:1: SyntaxWarning: invalid escape sequence '\p'

>>> x                                     # Keeps \ literally (and displays it as \\)
'C:\\py\\code'
>>> len(x)                               # But not for long: don't rely on this anymore!
10
```

Despite this behavior being expected (and even relied upon) for three decades, it has recently been judged bad and has started issuing a warning when used. In Python 3.12, it invokes a syntax warning that doesn't stop your program but clutters your output with nags. Worse, this will be treated as a programming-ending *error* in a future Python, so you should not use this going forward—and are expected to change all the code you've written in the past that does!

Even without this backward-incompatible Python change, though, strings that rely on this behavior seem as likely to lose backslashes in escapes as to retain them elsewhere. Instead, code literal backslashes explicitly such that they are retained in your strings in all Pythons, by either doubling them with `\\` (an escape for one `\`), or using raw strings—the topic of the next section.

Raw Strings Suppress Escapes

As we've already seen, escape sequences are handy for embedding special characters in strings. Sometimes, though, the special treatment of backslashes for introducing escapes can lead to trouble. It's surprisingly common, for instance, to see Python newcomers trying to open a file on Windows with a filename argument that looks something like this:

```
myfile = open('C:\new\text.dat', 'w')
```

thinking that they will open a file called *text.dat* in the directory *C:\new*. The problem with this is that `\n` is taken to stand for a newline character, and `\t` is replaced with a tab. In effect, the call tries to open a file named *C:(newline)ew(tab)ext.dat*, with usually less-than-stellar results.

This is just the sort of thing that *raw strings* are meant to address. If the letter *r* (uppercase or lowercase, but usually the latter) appears just before the opening quote of any string literal covered in this chapter, it turns off the escape mechanism. The result is that Python retains your backslashes *literally*, exactly as they appear in the string. Hence, to fix the filename problem, just remember to add the letter *r* on Windows:

```
myfile = open(r'C:\new\text.dat', 'w')           # Works: disable | escapes
```

Alternatively, because, as noted in the preceding section, two backslashes are really an escape sequence for one backslash, you can keep your backslashes by simply doubling them up when they should be taken verbatim:

```
myfile = open('C:\\new\\text.dat', 'w')          # Also works: || means |
```

In fact, Python itself sometimes uses this doubling scheme when it prints strings with embedded backslashes:

```
>>> path = r'C:\new\text.dat'                   # Raw string: keep |s
>>> path                                         # Show as Python code
'C:\\new\\text.dat'
>>> print(path)                                 # User-friendly format
C:\new\text.dat
>>> len(path)                                   # String length: |s
15
```

As covered in [Chapter 5](#), the default format at the interactive prompt prints results as if they were code, and therefore escapes backslashes in the output. The `print` statement provides a more user-friendly format that shows that there is actually only one backslash in each spot. To verify this is the case, you can check the result of the built-in `len` function, which returns the number of characters in the string, independent of display formats. If you count the characters in the `print(path)` output, you'll see that there really is just 1 character per backslash, for a total of 15.

Besides directory paths on Windows, raw strings are also commonly used for *regular expressions* in text pattern matching, supported with the `re` module introduced in [Chapter 37](#). Also note that Python scripts can usually use *forward slashes* in directory paths on both Windows and Unix because this slash is interpreted portably (e.g., `'C:/new/text.dat'` works when opening Windows files, too). Raw strings are useful for paths using native Windows backslashes, though, and any other time you want to ensure that Python will leave your `\` alone. They also work for triple-quoted strings to suppress escapes (and future invalid-escape errors!) in text of the sort up next.



Raw-string quirk: Despite its role, even a raw string *cannot end* in a single backslash, because the backslash escapes the following quote character. That is, `r'...\'` is not a valid string literal: you still must escape the surrounding quote character to embed it in the string, and Python assumes that is your intent. The upshot is that a raw string cannot end in an odd number of backslashes, including one. If you need to end a raw string with a single backslash, you can use two and slice off the second (`r'...\['[:-1]`), tack one on manually (`r'...' + '\\'`), or skip the raw string syntax and just double up the backslashes in a normal string (`'...\\'`). All three of these forms create the same two-character string with a Unicode ellipsis and one backslash.

Triple Quotes and Multiline Strings

So far, you've seen single quotes, double quotes, escapes, and raw strings in action. Python also has a triple-quoted string literal format, sometimes called a *block string*, that is a syntactic convenience for coding multiline data. This form begins with three quotes (of either the single or double variety), is followed by any number of lines of text, and is closed with the same triple-quote sequence that opened it. Single and double quotes embedded in the string's text may be, but do not have to be, escaped—the string does not end until Python sees three unescaped quotes of the same kind used to start the literal. For example:

```
>>> quip = """Python strings
...     sure have
...     a lot of options"""
>>>
>>> quip
'Python strings\n sure have\na lot of options'
```

This string spans three lines. As you learned in [Chapter 3](#), the interactive prompt changes to `...` on continuation lines like this in some interfaces, but not in others. This book omits the dots in some examples to enable cut-and-paste, but don't type them yourself if they're listed as they are here but absent in your REPL, and extrapolate as needed.

Prompts aside, Python collects all the triple-quoted text in this example into a single multiline string, with embedded newline characters (`\n`) at the places where your code has physical line breaks. Notice that, as in the literal, the second line in the result has leading spaces, but the third does not—what you type is truly what you get. To see the string with the newlines interpreted, print it instead of echoing:

```
>>> print(quip)
Python strings
    sure have
a lot of options
```

In fact, triple-quoted strings will retain all the enclosed text, *including* any to the right of your code that you might intend as *comments*. So don't do this—put your comments above or below the quoted text, or use the automatic concatenation of adjacent strings mentioned earlier, with explicit newlines if needed, and surrounding parentheses to allow line spans (you'll learn more about this latter form when you study syntax rules in [Chapters 10](#) and [12](#)):

```
>>> menu = """
... Open          # Comments here added to string!
... Save          # Ditto
... """
>>> menu
'\nOpen          # Comments here added to string!\nSave          # Ditto\n'

>>> menu = (
... 'Open\n'      # Comments here ignored
... 'Save\n'      # But newlines not automatic
... )
>>> menu
'Open\nSave\n'
```

So why use triple-quoted strings? For one thing, they are useful anytime you need *multiline text* in your program—for example, to embed multiline error messages, or HTML, XML, JSON, or YAML code in your Python source code files. You can usually embed such blocks directly in your scripts by triple-quoting without resorting to external text files or concatenation and newline characters.

Triple-quoted strings are also commonly used for *docstrings* (documentation strings), which are string literals that are taken as comments when they appear at specific points in your file (and are covered in full in [Chapter 15](#)). These don’t have to be triple-quoted blocks, but they usually are to allow for multiline comments and may need to be triple-quoted raw strings (e.g., `r'''`) to avoid invalid escape errors in the future (see the 3.12 syntax warnings noted previously).

Finally, triple-quoted strings are also sometimes used as a “horribly hackish” way to *temporarily disable* lines of code during development. Really, it’s not too horrible, and it’s actually a fairly common practice today, but it wasn’t the original intent of the literal. If you wish to turn off a few lines of code and run your script again, simply put three quotes above and below them, like this:

```
X = 1
"""
import os                                # Disable this code temporarily
print(os.getcwd())
"""
Y = 2
```

This was tagged as “hackish” because Python really might make a string out of the lines of code disabled this way, but this is probably not significant in terms of performance. For large sections of code, it’s also easier than manually adding hash marks before each line and later removing them. This is especially true if you are using a text editor that does not have support for editing Python code specifically. In Python, practicality often beats aesthetics.

Strings in Action

Once you’ve created a string with the literal expressions we just met, you will almost certainly want to do things with it. This section and the next two demonstrate string expressions, methods, and formatting—the first line of text-processing tools in the Python language.

Basic Operations

Let’s begin by interacting with the Python interpreter to illustrate the basic string operations listed earlier in [Table 7-1](#). You can concatenate strings using the `+` operator and repeat them using the `*` operator:

```

>>> len('abc')           # Length: number of items
3
>>> 'abc' + 'def'        # Concatenation: a new string
'abcdef'
>>> 'Py!' * 4            # Repetition: like 'Py!' + 'Py!' + 'Py!' + 'Py!'
'Py!Py!Py!Py!'

```

The `len` built-in function here returns the length of a string (or any other object with a length). Formally, adding two string objects with `+` creates a new string object, with the contents of its operands joined, and repetition with `*` is like adding a string to itself a given number of times (minus one). In both cases, Python lets you create arbitrarily sized strings; there's no need to predeclare anything in Python, including the sizes of data structures—you simply build string objects as needed and let Python manage the underlying memory space automatically, as we learned in [Chapter 6](#).

Repetition may seem a bit obscure at first, but it comes in handy in a surprising number of contexts. For example, to print a line of 80 dashes, you can count up to 80, or let Python count for you:

```

>>> print('----- ...more... -----')    # 80 dashes, the hard way
>>> print('-' * 80)                        # 80 dashes, the easy way

```

Notice that the *operator overloading* and *polymorphism* called out in [Chapter 5](#) and earlier is at work here already: we're using the same `+` and `*` operators that perform addition and multiplication when using numbers. Python does the correct operation because it knows the types of the objects being added and multiplied. But be careful: the rules aren't quite as liberal as you might expect. For instance, Python doesn't allow you to mix numbers and strings in `+` expressions: `'abc'+9` raises an error instead of automatically converting 9 to a string (we'll fix this ahead).

As shown near the end of [Table 7-1](#), you can also iterate over strings in loops using `for` statements, which repeat actions, and test membership for both characters and substrings with the `in` expression operator, which is essentially a search. For substrings, `in` is much like the `str.find()` method covered later in this chapter, but it returns a Boolean result instead of the substring's position (don't be alarmed if the following's `print` indents your prompt; its `end=' '` changes the default newline character at the end of the display to a space):

```

>>> myjob = 'hacker'
>>> for c in myjob:           # Step through items, print each + ' '
...     print(c, end=' ')    # Suppress newlines after each item
...
h a c k e r
>>> 'k' in myjob              # Found
True
>>> 'z' in myjob              # Not found
False
>>> 'HACK' in 'abcHACKdef'    # Substring search, no position returned
True

```

The `for` loop, previewed in [Chapter 4](#), assigns a variable to successive items in a sequence (here, a string) and executes one or more statements (normally indented) for each item. In effect, the variable `c` becomes a cursor stepping across the string's characters. Because iteration turns out to be a big idea in Python, we will discuss iteration tools like these and others listed in [Table 7-1](#) in more detail later in this book (see [Chapters 14](#) and [20](#)).

Indexing and Slicing

Because strings are ordered collections (a.k.a. *sequences*) of characters, we can access their components by position. As introduced in [Chapter 4](#), characters in a string are fetched by *indexing*—providing the numeric offset of the desired component in square brackets after the string. You get back the one-character string at the specified position.

As in most C-like languages, Python offsets start at 0 and end at one less than the length of the string (and the “start at 0” part may be a short-lived hurdle if you’re accustomed to counting from 1). Unlike C, however, Python also lets you fetch items from sequences such as strings using *negative* offsets. Technically, a negative offset is added to the length of a string to derive a positive offset, but you can also think of negative offsets as counting backward from the end. The following interaction demonstrates:

```
>>> S = 'code'
>>> S[0], S[-2]                # Indexing from front or end
('c', 'd')
>>> S[1:3], S[1:], S[:-1]      # Slicing: extract a section
('od', 'ode', 'cod')
```

In this code, the first line defines a four-character string and assigns it to the name `S`. The next line *indexes* it in two ways: `S[0]` fetches the item at offset 0 from the left—the one-character string `'c'` at the front; and `S[-2]` gets the item at offset 2 back from the end—or equivalently, at offset $(4 + (-2))$ from the front.

The last line in the foregoing example demonstrates *slicing*, a generalized form of indexing that returns an entire *section*, instead of a single item. It can be used to extract columns of data, chop off prefixes and suffixes, and more. Slicing can also be viewed as a type of *parsing* (decomposing content), especially when applied to strings, because it’s an easy way to extract substrings. In fact, we’ll explore slicing in the context of text parsing later in this chapter.

Slicing works like this: when you index a sequence object such as a string on a pair of offsets separated by a colon, Python returns a new object containing the contiguous section identified by the offset pair. The left offset is taken to be the lower bound (*inclusive*), and the right is the upper bound (*noninclusive*). That is, Python fetches all items from the lower bound up to but not including the upper bound and returns a new object containing the fetched items. If omitted, the left and right bounds default to 0 and the length of the object you are slicing, respectively.

For instance, in the example we just ran, `S[1:3]` extracts the items at offsets 1 and 2—it grabs the second and third items and stops before the fourth item at offset 3. Next, `S[1:]` gets *all items beyond the first*—the upper bound, which is not specified, defaults to the length of the string, which is off the end. Finally, `S[:-1]` fetches *all but the last item*—the lower bound defaults to 0, and `-1` refers to the last item, noninclusive. In more graphic terms, indexes and slices map to cells as shown in [Figure 7-1](#).

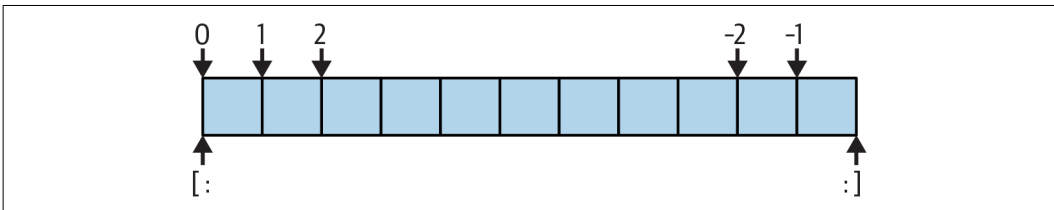


Figure 7-1. Indexes and slices: positives start from the left (0) and negatives from the right (−1)

All of which may seem confusing at first glance, but indexing and slicing are simple and powerful tools to use once you get the knack. Remember, if you're unsure about the effects of a slice, try it out interactively. In the next chapter, you'll see that it's even possible to change an entire section of another object in one step by *assigning* to a slice (though not for immutables like the strings we're studying here). For now, here's a cheat sheet of the details for reference:

Indexing—`S[I]`—fetches components at offsets in sequences:

- The first item is at offset 0.
- Negative indexes mean counting backward from the end or right.
- Out-of-bounds offsets are an error.
- `S[0]` fetches the first item.
- `S[-2]` fetches the second item from the end (like `S[len(S)-2]`).

Slicing—`S[I:J]`—extracts contiguous sections of sequences:

- The upper bound is noninclusive.
- Slice bounds default to 0 and the sequence length, if omitted.
- Out-of-bounds offsets are adjusted to be in bounds.
- `S[1:3]` fetches items at offsets 1 up to but not including 3.
- `S[1:]` fetches items at offset 1 through the end (the sequence length).
- `S[:3]` fetches items at offset 0 up to but not including 3.
- `S[:-1]` fetches items at offset 0 up to but not including the last item.
- `S[:]` fetches items at offsets 0 through the end—making a top-level copy of `S`.

Extended slicing—`S[I:J:K]`—accepts a step (or stride) `K`, which defaults to `+1`:

- Allows for skipping items and reversing order—see the next section.

The second-to-last bullet item listed here turns out to be a common technique: `S[:]` makes a full top-level *copy* of a sequence object—an object with the same value, but a distinct piece of memory (you'll find more on copies in [Chapter 9](#)). This isn't very useful for immutable objects like strings, but it comes in handy for objects that may be changed in place, such as lists: making a copy can avoid the side effects of shared references shown in [Chapter 6](#).

Also per the cheat sheet, slices differ from indexes in their policy on *out-of-bounds* (off the end) offsets: they're always *errors* in indexing, because the offset does not exist, but *scaled* to be in bounds in slicing, because this can be useful in programs that need to accommodate sizes flexibly:

```
>>> S = 'code'
>>> S[99]
IndexError: string index out of range
>>> S[1:99]
'ode'
```

Because we're going to explore this oddity in an end-of-part exercise, though, we'll cut the story short here.

Extended slicing: The third limit and slice objects

Though not commonly used, slice expressions also support an optional third index, used as a *step* (sometimes called a *stride*). The step is added to the index of each item extracted. With it, the full-blown form of a slice is `S[I:J:K]`, which means “extract all the items in `S`, from offset `I` through `J`–1, by `K`.” The third limit, `K`, defaults to `+1`, which is why normally all items in a slice are extracted from left to right. If you specify an explicit value, however, you can use the third limit to skip items or to reverse their order.

For instance, `S[1:10:2]` will fetch *every other item* in `S` from offsets 1–9; that is, it will collect the items at offsets 1, 3, 5, 7, and 9. As usual, the first and second limits default to 0 and the length of the sequence, respectively, so `X[:2]` gets every other item from the beginning to the end of the sequence:

```
>>> S = 'abcdefghijklmnop'
>>> S[1:10:2]                # Skipping items
'bdfhj'
>>> S[:2]
'acegikmo'
```

You can also use a negative stride to collect items in the opposite order. For example, in the slicing expression `S[::-1]`, the first two bounds default to sequence length–1 and –1 (they really default to `None` and `None`, but that’s unimportant here), and a stride of –1 indicates that the slice should go from right to left instead of the usual left to right. In much simpler terms, the effect is to *reverse* the sequence:

```
>>> S = 'hello'
>>> S[::-1]                  # Reversing items
'olleh'
```

With a negative stride, the meanings of the first two bounds are essentially reversed. That is, the slice `S[5:1:-1]` fetches the items from 2 to 5, in reverse order (the result contains items from offsets 5, 4, 3, and 2):

```
>>> S = 'abcdefg'
>>> S[5:1:-1]                # Bounds roles differ
'fdec'
```

Skipping and reversing like this are the most common use cases for three-limit slices, but see Python’s standard-library manual for more details (or run a few experiments interactively). We’ll revisit three-limit slices again later in this book, in conjunction with the `for` loop statement.

Later in the book, you’ll also learn that slicing is equivalent to indexing with a *slice object*, a finding of importance to class writers seeking to support both operations:

```
>>> 'code'[1:3]              # Slicing syntax
'od'
>>> 'code'[slice(1, 3)]      # Slice objects with index syntax + object
'od'
>>> 'code'[::-1]
'edoc'
>>> 'code'[slice(None, None, -1)]
'edoc'
```


Why You Will Care: Slices

Throughout this book, you'll meet common use-case sidebars such as this one that give you a peek at how some of the language features being discussed are typically used in real programs. Because you won't be able to make much sense of realistic use cases until you've seen more of the Python picture, these sidebars necessarily contain many references to topics not introduced yet; at most, you should consider them previews of ways that you may find these abstract language concepts useful for practical programming tasks.

For instance, you'll see later that the argument words listed on a system command line used to launch a Python program are made available in the `argv` attribute of the built-in `sys` module:

```
# File echo.py
import sys
print(sys.argv)

$ python3 echo.py -a -b -c
['echo.py', '-a', '-b', '-c']
```

Usually, you're interested only in inspecting the arguments that follow the program name. This leads to a typical application of slices: a single slice expression can be used to return all but the first item of a list. Here, `sys.argv[1:]` returns the desired list, `['-a', '-b', '-c']`. You can then process this list without having to accommodate the program name at the front.

Slices are also often used to clean up lines read from input files, of the sort we'll study in [Chapter 9](#). If you know that a line will have a newline character at the end (a `\n`), you can get rid of it with a single expression such as `line[:-1]`, which extracts all but the last character in the line. In both cases, slices do the job of logic that must be explicit in a lower-level language.

Having said that, calling the `line.rstrip` method is often preferred for stripping newline characters because this call leaves the line intact if it has no newline character at the end—a common case for files created with some text-editing tools. Slicing works only if you're sure the line is properly terminated.

String Conversion Tools

One of Python's design mottos is that it refuses the temptation to guess. As a prime example, you cannot add a number and a string together in Python, even if the string looks like a number (i.e., is all digits):

```
>>> '62' + 1
TypeError: can only concatenate str (not "int") to str
```

This is by design: because `+` can mean both addition and concatenation, the choice of conversion would be ambiguous—do you want `'621'` or `63`? Instead, Python treats this as an error. In Python, magic is generally omitted if it will make your coding life more complex.

What to do, then, if your script obtains a number as a text string from a file or user interface? The trick is that you must simply employ conversion tools before you can treat a string like a number, or vice versa. For instance:

```
>>> int('62'), str(62)           # Convert from/to string
(62, '62')
```

The `int` function converts a string to a number, and the `str` function converts a number to its string representation (essentially, what it looks like when printed). Now, although you can't mix strings and number types around operators such as `+`, you can manually convert operands before that operation if needed:

```
>>> S = '62'
>>> I = 1
>>> S + I
TypeError: can only concatenate str (not "int") to str

>>> int(S) + I           # Force addition
63

>>> S + str(I)           # Force concatenation
'621'
```

Similar built-in functions handle floating-point-number conversions to and from strings, if you need to mix the two in expressions:

```
>>> float('1.5') + 2.8
4.3
>>> '1.5' + str(2.8)
'1.52.8'
```

The built-in `eval` function introduced in [Chapter 5](#) runs a string containing Python expression code, and so can also convert a string to any kind of object. The functions `int` and `float` convert only to numbers, but this restriction means they are usually faster (and more secure, because they do not accept arbitrary expression code). As we also saw briefly in [Chapter 5](#), string formatting provides other ways to convert numbers to strings; more on it ahead.

Character-code conversions

On the subject of conversions, it is also possible to convert a single character to its underlying integer code by passing it to the built-in `ord` function—this returns the numeric “ordinal” value used to represent the corresponding character in memory (technically, its Unicode *code point*, as you'll learn in [Chapter 37](#), but this isn't crucial yet). The `chr` function performs the inverse operation, taking an integer code and converting it to the corresponding character:

```
>>> ord('h')             # Character => ID (code point)
104
>>> chr(104)             # ID => character (string)
'h'

>>> for c in 'hack':     # All code points in a string
...     print(c, ord(c))
...
h 104
a 97
c 99
k 107
```

You can use a loop to apply `ord` to all characters in a string as shown, but these tools can also be used to perform a simple sort of string-based math. To advance to the next character, for example, convert and do the math in integer:

```
>>> S = '5'
>>> S = chr(ord(S) + 1)
>>> S
'6'
>>> S = chr(ord(S) + 1)
>>> S
'7'
```

At least for single-character strings, this provides an alternative to using the built-in `int` function to convert from string to integer (though this only makes sense if character ordinals are ordered as your code expects!):

```
>>> int('5')
5
>>> ord('5') - ord('0')
5
```

String comparisons

Another reason for introducing ordinals here is that it helps us understand *string comparisons*: when we compare two text strings, Python automatically compares them left to right, character by character, and lexicographically—that is, by the same character code-point values returned by `ord`—until the first mismatch or end of either string. In the following, for example, the code point of `t` is greater than that of `k`, and the longer string at the end wins:

```
>>> 'hack' == 'hack', 'hact' > 'hack', 'hacker' > 'hack'
(True, True, True)
```

The same holds true for the byte strings you’ll meet in [Chapter 37](#) (they are compared byte for byte until a result is known), and the next chapter’s richer collections like lists do similar (Python compares all their parts for you).

“Changing” Strings Part 1: Sequence Operations

Remember the term *immutable sequence*? As we’ve seen, being a *sequence* means that strings support operations like concatenation, repetition, indexing, and slicing. The *immutable* part means that you cannot change a string in place—for instance, by assigning to an index:

```
>>> S = 'text'
>>> S[0] = 'n'
TypeError: 'str' object does not support item assignment
```

How to modify textual information in Python, then? To change a string, you generally need to build a *new* string using tools such as concatenation and slicing, and assign the result back to the string’s original name if desired:

```
>>> S = 'text'
>>> S = S + 'ual!'          # To change a string, make a new one
>>> S
'textual!'
>>> S = S[:4] + ' processing' + S[-1]
>>> S
'text processing!'
```

The first example adds a substring at the end of `S`, by concatenation. Really, it makes a *new string* and assigns it back to `S` to save it, but you can think of this as “changing” the original string. The second

example replaces three characters with many by slicing, indexing, and concatenating. As you'll see in the next section, you can achieve similar effects with string methods like `replace`:

```
>>> S = 'text'
>>> S = S.replace('ex', 'hough')
>>> S
'thought'
```

Like every operation that yields a new string value, string methods generate new string objects. If you want to retain those objects, you can assign them to variable names. Whether by sequence operations or methods, generating a new string object for each string change is not as inefficient as it may sound—remember, as discussed in the preceding chapter, Python automatically garbage-collects (reclaims the space of) old unused string objects as you go, so newer objects reuse the space held by prior values. Python is usually more efficient than you might expect.

But string methods can do much more, as the next section will explain.



Except for bytearray: As previewed in [Chapter 4](#) and to be covered in [Chapter 37](#), Python has a string type known as `bytearray`, which is mutable and so may be changed in place. `bytearray` objects aren't really text strings; they're sequences of small, 8-bit integers. However, they support most of the same operations as normal strings and print as ASCII characters when displayed. Accordingly, they provide another option for large amounts of simple 8-bit text that must be changed frequently. Richer Unicode text and *str* strings in general, though, require techniques shown here.

String Methods

In addition to all the string operations already introduced, strings provide a set of *methods* that support more sophisticated text-processing goals. In Python, expressions and built-in functions may work across a range of types, but methods are generally *specific to object types*—string methods, for example, work only on string objects. Some method names are used by multiple objects in Python for consistency (e.g., many have `count` and `copy` methods, and most mutables have a `pop`), but they are still more type specific than other tools.

Method Call Syntax

As introduced in [Chapter 4](#), methods are simply functions that are associated with and act upon particular objects. Technically, they are attributes attached to objects that happen to reference callable functions which always have an *implied subject*. In finer-grained detail, functions are packages of code, and method calls combine two operations at once—an attribute fetch and a call:

Attribute fetches

An expression of the form `object.attribute` means “fetch the value of *attribute* in *object*.”

Call expressions

An expression of the form `function(arguments)` means “invoke the code of *function*, passing zero or more comma-separated *argument* objects to it, and return *function*'s result value.”

Putting these two together allows us to call a method of an object. The method call expression:

```
object.method(arguments)
```

is evaluated from left to right—Python will first fetch the *method* of the *object* and then call it, passing in both *object* and the *arguments*. Or, in plain words, the method call expression means this:

Call *method* to process *object* with *arguments*.

If the method computes a result, it will also come back as the result of the entire method-call expression. As a more tangible example:

```
>>> S = 'hack'
>>> result = S.find('ac')      # Call the find method to look for 'ac' in string S
```

This mapping holds true for methods of both built-in types, as well as user-defined classes we'll study later. As you'll see throughout this part of the book, most objects have callable methods, and all are accessed using this same method-call syntax. To call an object method, as you'll see in the following sections, you have to go through an existing object; methods cannot be run (and make little sense) without a subject.

All String Methods (Today)

Table 7-3 summarizes the methods and call patterns for built-in string objects in Python 3.12. These change over time, so be sure to check Python's standard-library manual for the most up-to-date list, or run a `dir` or `help` call interactively on any string or the `str` type name, as shown in [Chapter 4](#).

In this table, `S` is a string object; optional arguments are enclosed in `[]` brackets; nested `[]` mean optional following an optional; `*X` and `**X` mean any number `X`; and the listed methods implement higher-level operations such as splitting and joining, case conversions, content tests, and substring searches and replacements.

Table 7-3. String method calls in Python 3.12

<code>S.capitalize()</code>	<code>S.ljust(width [, fill])</code>
<code>S.casefold()</code>	<code>S.lower()</code>
<code>S.center(width [, fill])</code>	<code>S.lstrip([chars])</code>
<code>S.count(sub [, start [, end]])</code>	<code>S.maketrans(x [, y [, z]])</code>
<code>S.encode([encoding [, errors]])</code>	<code>S.partition(sep)</code>
<code>S.endswith(suffix [, start [, end]])</code>	<code>S.removeprefix(prefix)</code>
<code>S.expandtabs([tabsize])</code>	<code>S.removesuffix(suffix)</code>
<code>S.find(sub [, start [, end]])</code>	<code>S.replace(old, new [, count])</code>
<code>S.format(fmtstr, *args, **kwargs)</code>	<code>S.rfind(sub [, start [, end]])</code>
<code>S.format_map(mapping)</code>	<code>S.rindex(sub [, start [, end]])</code>
<code>S.index(sub [, start [, end]])</code>	<code>S.rjust(width [, fill])</code>
<code>S.isalnum()</code>	<code>S.rpartition(sep)</code>
<code>S.isalpha()</code>	<code>S.rsplit([sep [, maxsplit]])</code>
<code>S.isascii()</code>	<code>S.rstrip([chars])</code>
<code>S.isdecimal()</code>	<code>S.split([sep [, maxsplit]])</code>
<code>S.isdigit()</code>	<code>S.splitlines([keepends])</code>

<code>S.isidentifier()</code>	<code>S.startswith(prefix [, start [, end]])</code>
<code>S.islower()</code>	<code>S.strip([chars])</code>
<code>S.isnumeric()</code>	<code>S.swapcase()</code>
<code>S.isprintable()</code>	<code>S.title()</code>
<code>S.isspace()</code>	<code>S.translate(map)</code>
<code>S.istitle()</code>	<code>S.upper()</code>
<code>S.isupper()</code>	<code>S.zfill(width)</code>
<code>S.join(iterable)</code>	

As you can see, strings have many methods, and we don't have space to cover them all here; omissions can be found in other resources when needed. To help you get started, though, let's work through some code that demonstrates some of the most commonly used methods in action and illustrates Python text-processing basics along the way.

“Changing” Strings, Part 2: String Methods

As we've seen, most strings cannot be changed in place directly because they are immutable. We explored changing strings with sequence operations in the preceding section, but let's resume that story here in the context of methods.

By way of review, to make a new text value from an existing string, you can construct a new string with sequence operations such as slicing and concatenation. For example, to replace two characters in the middle of a string, you can use code like this, much as we did in the prior section:

```
>>> S = 'textly!'
>>> S[:4] + 'ful' + S[-1]           # Make a new string with sequence ops
'textful!'
```

But, if you're really just out to replace a substring, you can use the string `replace` method instead:

```
>>> S = 'textly!'
>>> S.replace('ly', 'ful')          # Replace all 'ly' with 'ful' in S
'textful!'
```

The `replace` method is more general than this code implies. It takes as arguments the original substring (of any length) and a new substring (of any length) to replace the original, and performs a global search and replace—subject to an optional third argument that limits the number of replacements made:

```
>>> '---@---@---'.replace('@', 'PY', 2)
'--PY--PY--@--'
```

In such a role, `replace` can be used as a tool to implement simple *template* replacements (e.g., in form letters). If you need to replace one fixed-size string that can occur at any offset, you can do a replacement again, or search for the substring with the string `find` method and then slice:

```
>>> S = 'xxxxPYxxxxPYxxxx'
>>> where = S.find('PY')           # Search for position
>>> where                                # Occurs at offset 4
4
>>> S = S[:where] + 'CODE' + S[(where+2):]
>>> S
'xxxxCODExxxxPYxxxx'
```

The `find` method returns the offset where a substring appears, or `-1` if it is not found (it searches from the front by default, and its cousin `rfind` searches in reverse). As we saw earlier, this is a *substring search* operation just like the `in` expression, but `find` returns the position of a located substring. In this context, `replace` does the job easier, and can do more—in the following, replacing both multiple occurrences and multiple targets:

```
>>> S = 'xxxPYxxxPYxxx'
>>> S.replace('PY', 'CODE', 1)          # Replace one
'xxxCODExxxPYxxx'

>>> S.replace('PY', 'CODE')            # Replace all
'xxxCODExxxCODExxx'

>>> 'xxxWHATxxxHOWxxx'.replace('WHAT', 'CODE').replace('HOW', 'PYTHON')
'xxxCODExxxPYTHONxxx'
```

As a reminder, `replace` returns a new string object each time (which is why two calls can be strung together here). Because strings are immutable, methods, like sequence operations, never really change the subject string in place—even if they are called “replace”! To save the new string object produced by a method call, assign it to a name:

```
>>> S = S.replace('PY', 'CODE')
>>> S
'xxxCODExxxCODExxx'
```

The fact that concatenation operations and the `replace` method generate new string objects each time they are run is a potential downside of using them to change strings: each interim result must create a full-fledged object with a fresh copy of its text. If you have to apply many changes to a very large string, you might be able to improve your script’s performance by converting the string to an object that does support in-place changes:

```
>>> S = 'text'
>>> L = list(S)                      # Explode string into a list
>>> L
['t', 'e', 'x', 't']
```

The built-in `list` function (really, an object construction call) builds a new list out of the items in any sequence (or other iterable)—in this case, “exploding” the characters of a string into a list. Once the string is in this form, you can make multiple changes to it without generating a new copy for each change:

```
>>> L[0] = 'h'                      # Works for lists, not strings
>>> L[3] = '!'
>>> L
['h', 'e', 'x', '!']
```

After your changes, you can convert back to a string if needed (e.g., to write to a file) by using the string `join` method to “implode” the list back into a string:

```
>>> S = ''.join(L)                  # Implode back to a string
>>> S
'hex!'
```

The `join` method may look a bit backward on first encounter. Because it is a method of strings (not of lists), it is called through the desired *delimiter* string. `join` puts the strings in a list (or other iterable) together, with the delimiter between list items; in this case, it uses an empty string delimiter to convert from a list back to a string. More generally, any string delimiter and iterable of strings will do:

```
>>> 'PY'.join(['which', 'language', 'is', 'best', '?'])
'whichPYlanguagePYisPYbestPY?'
```

Though subject to Python implementation, joining substrings all at once might run faster than concatenating them individually. The mutable `bytearray` string noted earlier may help with efficiency too; because it can be changed in place, it offers an alternative to this `list/join` combo for simple kinds of byte-sized text like ASCII.

More String Methods: Parsing Text

Another common role for string methods is as a simple form of text *parsing*—that is, analyzing structure and extracting substrings. To extract substrings at fixed offsets, we can employ *slicing* techniques:

```
>>> line = 'aaa bbb ccc'

>>> col1 = line[:3]
>>> col2 = line[4:8]
>>> col3 = line[-3:]

>>> col1, col2, col3
('aaa', 'bbb ', 'ccc')
```

Here, the columns of data appear at fixed offsets and so may be sliced out of the original string. This technique passes for parsing, as long as the components of your data have known positions. If instead some sort of delimiter separates the data, you can pull out its components by *splitting*. This will work even if the data may show up at arbitrary positions within the string:

```
>>> line = 'aaa bbb      ccc'
>>> cols = line.split()
>>> cols
['aaa', 'bbb', 'ccc']
```

The string `split` method chops up a string into a list of substrings, around a delimiter string. We didn't pass a delimiter in the prior example, so it defaults to whitespace—the string is split at groups of one or more spaces, tabs, and newlines, and we get back a list of the resulting substrings. In other applications, more tangible delimiters may separate the data. To demo, the next example splits (and hence parses) the string at commas, a separator common in some database roles (string conversion tools covered earlier can change substrings here into numbers):

```
>>> line = 'Python,3.12,scripting,33'
>>> line.split(',')
['Python', '3.12', 'scripting', '33']
```

Delimiters can be longer than a single character, too:

```
>>> line = 'youPYarePYaPYstringPYcoder'
>>> line.split('PY')
['you', 'are', 'a', 'string', 'coder']
```

Although there are limits to the parsing potential of slicing and splitting, both run fast and can handle basic text-extraction chores. Comma-separated text data is also part of the CSV file format; for a more advanced tool on this front, see also the `csv` module in Python's standard library.

Other Common String Methods

Other string methods have more focused purposes—for example, to strip off whitespace at the end of a line of text, perform case conversions, test content, and test for a substring at the end or front:

```
>>> line = "Python's strings are awesome!\n"

>>> line.rstrip()           # Drop whitespace (or other)
"Python's strings are awesome!"
>>> line.upper()            # Case conversions
"PYTHON'S STRINGS ARE AWESOME!\n"
>>> line.isalpha()          # Content tests
False
>>> line.endswith('awesome!\n') # Suffix and prefix tests
True
>>> line.startswith('Python')
True
```

Alternative techniques can also sometimes be used to achieve the same results as string methods—the in membership operator can be used to test for the presence of a substring, for instance, and length and slicing operations can be used to mimic `endswith`:

```
>>> line.find('awesome') != -1      # Search via method call or expression
True
>>> 'awesome' in line
True

>>> sub = 'awesome!\n'
>>> line.endswith(sub)              # End test via method call or slice
True
>>> line[-len(sub):] == sub
True
```

Note that none of the string methods accepts *patterns*—for pattern-based text processing, you must use the Python `re` standard-library module, an advanced tool that will be introduced briefly in [Chapter 37](#) but is mostly outside the scope of this text. Because of their limitations, though, string methods may run more quickly than the `re` module's tools.

Again, because there are so many methods available for strings, we won't look at every one here. You'll see some additional string examples later in this book, but for more details you can also turn to the Python library manual and other reference resources, or simply experiment interactively on your own. As noted in [Chapter 4](#), `help(S.method)` gives info for a *method* of any string object `S`; use `help(str.method)` if you have no `S`.

All that being said, one method is noticeably absent from this section's coverage: `format` performs string formatting, which combines many operations in a single step. It's also part of a larger topic in Python, which we turn to next.

String Formatting: The Triathlon

Although you can get a lot done with the string methods and sequence operations you've already met, Python also provides a more advanced way to combine string processing tasks: *string formatting* allows us to perform multiple type-specific substitutions on a string in a single step. It's never strictly required, but it can be convenient, especially when laying out text to be displayed to a program's users.

We’ve used string formatting informally in this book already, but it’s finally time to dig into its details. As suggested in earlier examples, this story is regrettably convoluted by Python’s history: there are today *three different string-formatting tools* that broadly overlap in functionality. This curious state of affairs reflects a common pattern in software development—new tools arise that boldly promise to be radical improvements over the past, only to be supplanted by even newer tools that boldly make the exact same claims.

The net effect handicaps languages with redundancy, and users with unnecessarily steep learning curves. While learning resources could present just one of many options, that would both impose authors’ opinions and do a vast disservice to readers: even if you’re able to pick just one of the formatting tools for your own work, the fact that the others have been available for decades and have been used by millions of programmers virtually guarantees that you’ll be seeing them in the wild when you begin reusing other people’s code. Try as it may, the new cannot erase the old.

Hence, this chapter presents all three formatting tools for the sake of inclusiveness. If you’re new to Python or programming in general, you probably should focus on the current latest-and-greatest *f-string* option—not because it’s necessarily “better,” but because it’s more likely to garner development attention and less likely to be deprecated in a backward-incompatible future (its predecessors have been spared this fate to date, but Python has a history here).

But that’s not to say that the others are out of the race: the expression and method alternatives may feel more comfortable to readers with backgrounds in some other tools, and both are pervasive in the vast reams of Python code written over the last thirty-some years. Learning all three options hedges your bets best.

String-Formatting Options

As a preview of what we’re going to explore in this section, here are today’s entries in the string-formatting race:

Formatting expression: `'...%s...%s...' % (value, value)`

The original technique available since Python’s inception, this form is loosely based upon the C language’s `printf` model and sees widespread use in much existing code. Values on the right replace targets on the left.

Formatting method: `'...{}...{}...' .format(value, value)`

A newer technique added in Python 3.0, this form is derived in part from a same-named tool in C#/.NET. It largely overlaps with the expression’s functionality but aims to address usage modes subjectively deemed subpar.

Formatting literal: `f'...{value}...{value}...'`

The very latest (so far) added in Python 3.6, this form is known as *f-strings*. It shares much with the method but apes a host of languages that support *string interpolation*—substituting inline expressions with their results.

You can also format strings manually with string methods, though it’s too cumbersome to count. And technically, an additional tool, `string.Template`, predates the method—and drives the formatting set’s length up to a whopping *four*—but it’s so scantily used that it gets less billing than the three primary options above and is relegated to a brief sidebar here (and you’d be excused for pretending it doesn’t exist at all, given the heft of the formatting toolbox!).

The following sections present all three formatting options above in turn. While it may be tempting to jump straight to whatever the blogosphere may be recommending as you read these words, these

sections partly build on each other (e.g., f-strings use the method's format specifier and assume its earlier coverage), so a linear read is suggested.

The String-Formatting Expression

Since string-formatting *expressions* are the original in this department, we'll start with them. Python defines the % binary operator to work on strings. You may recall that this is also the remainder of division, or modulus, operator for numbers. When applied to strings, the % operator provides a simple way to format values as strings according to a format definition. It's a much more concise way to code multiple substitutions than processing parts individually.

Formatting expression basics

To format strings with an expression:

1. On the *left* of the % operator, provide a format string containing one or more embedded conversion targets, each of which starts with a % (e.g., %d).
2. On the *right* of the % operator, provide the object that you want Python to insert into the format string on the left in place of the conversion target; for multiple targets, provide multiple objects in a tuple.

For instance, in the following formatting example, the integer 3 replaces the %d in the format string on the left, and the string 'format' replaces the %s. The result is a new string that reflects these two substitutions, which may be printed or saved for use in other roles:

```
>>> 'There are %d ways to %s!' % (3, 'format')      # Formatting expression
'There are 3 ways to format!'
```

Technically speaking, string formatting in any flavor is usually optional—you can generally do similar work with multiple concatenations and conversions. However, formatting allows us to combine many steps into a single operation. It's powerful enough to warrant a few more introductory examples:

```
>>> option = 'expression'
>>> 'Meet the formatting %s!' % option              # String substitution
'Meet the formatting expression!'

>>> '%d %s %g you' % (1, 'formatter', 4.0)         # Type-specific substitutions
'1 formatter 4 you'

>>> '%s -- %s -- %s' % (42, 3.14159, [1, 2, 3])    # All types match a %s target
'42 -- 3.14159 -- [1, 2, 3]'
```

The first example here plugs a string into the target on the left, replacing the %s marker. In the second example, three values are inserted into the target string.

Notice that when you're inserting more than one value, you need to group the values on the right in parentheses—that is, put them in a *tuple*. The % operator's right side generally expects a tuple of one or more items (or a dictionary of items for key references, covered ahead), but allows a single nontuple item if there is just one substitution target. You'll know which form to use when coding the expression, of course, but this difference was nevertheless deemed a quirk sufficient to justify other formatting options over time.

The third example again inserts three values—an integer, a floating-point number, and a list—but notice that all of the targets on the left are %s, which stands for conversion to string. As every type of

object can be converted to a string (the one used when printing), every object type works with the `%s` conversion code. Because of this, unless you need to do special formatting, `%s` is often the only code you need to remember for the formatting expression.

Again, keep in mind that formatting always makes a new string, rather than changing the string on the left; because strings are immutable, it must work this way. As before, assign the result to a variable name if you need to retain it.

Formatting expression custom formats

For more advanced type-specific formatting, you can use any of the conversion type codes listed in [Table 7-4](#) in formatting expressions; they appear after the `%` character in substitution targets. C programmers will recognize most of these because Python string formatting supports all the usual C `printf` format codes (but returns the result, instead of displaying it like `printf`). Some of the format codes in the table provide alternative ways to format the same type; for instance, `%e`, `%f`, and `%g` provide alternative ways to format floating-point numbers.

Table 7-4. Formatting-expression type codes

Code	Meaning
s	String (or any object's <code>str(X)</code> string)
r	Same as s, but uses <code>repr</code> , not <code>str</code>
a	Same as s, but uses <code>ascii</code> , not <code>str</code>
c	Character (integer code or string)
d	Decimal (signed base-10 integer)
i	Integer (see d)
u	Same as d (obsolete: no longer unsigned)
o	Octal integer (base 8)
x	Hex integer (base 16)
X	Same as x, but with uppercase letters
e	Floating point with exponent, lowercase
E	Same as e, but uses uppercase letters
f	Floating-point decimal
F	Same as f, but uses uppercase letters
g	Floating-point e or f
G	Floating-point E or F
%	Literal % (coded as <code>%%</code>)

All told, conversion targets in the format string on the expression's left side support a variety of conversion operations with a fairly sophisticated syntax all their own. In formal terms, the general structure of conversion targets looks like the following, where [...] denotes an optional part (its two square-bracket characters are not included in the expression's code), and no spaces are allowed between parts (though some parts may contain spaces):

```
%( (keyname) ) [ flags ] [ width ] [ . precision ] typecode
```

One of the *type code* characters in the first column of [Table 7-4](#) shows up at the end of this target string's format, at *typecode*. Between the % and this type code character, you can do any (or none) of the following:

- Provide a *key name* for indexing the dictionary used on the right side of the expression.
- List *flags* that specify zero padding (0), left justification (-), numeric sign (+), or a blank before positive numbers and a - for negatives (a space), where - overrides 0, and + overrides a space.
- Give a total but minimum field *width* for the substituted text.
- Set the number of digits (*precision*) to display after a decimal point for floating-point numbers.

Both the *width* and *precision* parts can also be coded as a * to specify that they should take their values dynamically from the next item in the input values on the expression's right side (useful when this isn't known until your code is run, but unavailable when *keynames* are used). And if you don't need any of these extra tools, a simple %s in the format string will be replaced by the corresponding value's default print string, regardless of its type.

Advanced formatting expression examples

Formatting target syntax is documented in full in the Python standard manuals and other reference resources, but to demonstrate common usage, let's explore a few examples. The first formats integers using the default, and then in a six-character field with left justification and zero padding:

```
>>> x = 1234
>>> res = 'integers: ...%d...%-6d...%06d' % (x, x, x)
>>> res
'integers: ...1234...1234 ...001234'
```

The %e, %f, and %g formats display floating-point numbers in different ways, as the following interaction demonstrates—%E is the same as %e but the exponent is uppercase, and g chooses formats by number content (it's formally defined to use exponential format e if the exponent is less than -4 or not less than precision, and decimal format f otherwise, with a default minimum total-digits precision of 6; no, really!):

```
>>> x = 1.23456789
>>> x
1.23456789
# Default REPL display

>>> '%e | %f | %g' % (x, x, x)
'1.234568e+00 | 1.234568 | 1.23457'

>>> '%E' % x
'1.234568E+00'
```

For floating-point numbers, you can achieve a variety of additional formatting effects by specifying left justification, zero padding, numeric signs, total field width, and digits after the decimal point. For simpler tasks, you might get by with simply converting to strings with a %s type code or the str built-in function we used earlier:

```
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23   | 01.23 | +001.2'

>>> '%s' % x, str(x)
('1.23456789', '1.23456789')
```

When sizes are not known until runtime, you can use a *dynamically* computed width and precision by specifying them with a `*` in the format string to force their values to be taken from the next item in the inputs to the right of the `%` operator—the 4 in the tuple here gives precision:

```
>>> '%f, %.2f, %.*f' % (1/3.0, 1/3.0, 4, 1/3.0)
'0.333333, 0.33, 0.3333'
```

As usual, experiment with some of these examples and operations on your own for more insight.

Dictionary-based formatting expressions

As a more advanced extension, `%` string formatting also allows conversion targets on the left to refer to the keys in a *dictionary* coded on the right and use the corresponding values. Syntactically, this form requires `()` key references on the left of `%`, and a single dictionary (or other mapping) on the right. Functionally, it allows formatting to be used as a basic *template* tool. You’ve met dictionaries only briefly thus far in [Chapter 4](#), but the following demos the idea:

```
>>> '%(qty)s more %(tool)s' % {'qty': 1, 'tool': 'formatter'}
'1 more formatter'
```

Here, the `(qty)` and `(tool)` in the format string on the left refer to *keys* in the dictionary literal on the right and fetch their associated values. Programs that generate text such as HTML or XML often use this form: they build up a dictionary of values and substitute them all at once with a single formatting expression, using key references and a template string either loaded from a file or coded in the script. The following demos the idea (notice that its first comment is above the triple quote to keep it out of the string, and “...” prompts may not appear in your REPL):

```
>>>                                     # Template with substitution targets
>>> reply = """
... Hello %(name)s!
... Welcome to %(year)s
... """

>>> values = {'name': 'Pat', 'year': 2024}    # Build up values to substitute
>>> print(reply % values)                    # Perform substitutions

Hello Pat!
Welcome to 2024
```

This trick is also sometimes used in conjunction with the `vars` built-in function, which returns a dictionary containing all the variables that exist in the place it is called:

```
>>> name = 'Pat'
>>> year = 2024
>>> vars()
{'name': 'Pat', 'year': 2024, ...plus built-in names set by Python...}
```

When used on the right side of a format operation, this allows the format string to refer to variables by *name*—using dictionary-key syntax:

```
>>> '%(name)s from %(year)s' % vars()        # Variables are keys in vars()
'Pat from 2024'
```

Although formatting expressions are positional by nature, dictionaries also allow them to *reuse values* more than once (see [Chapter 5](#) for another demo of this in action):

```
>>> '%(value)f, %(value).2f, %(value).f' % ({'value': 1 / 3.0})
'0.333333, 0.33, 0'
```

We'll study dictionaries in more depth in [Chapter 8](#). See also “Hex, Octal, and Binary” on [page 96](#) for examples that convert to hexadecimal and octal number strings with the `%x` and `%o` formatting expression target codes, which we won't repeat here. Additional formatting expression examples also appear ahead as comparisons to the formatting method and f-string—the first of which is this chapter's next topic.

The String-Formatting Method

As noted, Python 3.0 added a second way to format strings that some see as more Python specific. Unlike formatting expressions, the formatting method is not as closely based upon the C language's “printf” model; is sometimes more explicit in intent; and avoids the value-or-tuple quirk of `%`: substituted values are just arguments, whether one or many.

On the other hand, the new technique still relies on “printf” concepts like type codes and formatting specifications. Moreover, it largely overlaps with formatting expressions; often yields more verbose code; and in practice can be just as complex in most roles. And if we're all being honest, the value-or-tuple quirk of the `%` expression is much more of a concern in theory than practice. Luckily, the two are similar enough that many core concepts overlap.

Formatting method basics

The string object's `format` method at the heart of this option is based on function call syntax, instead of an expression. Specifically, it uses the call's subject string as a template and takes any number of arguments that represent values to be substituted according to the template.

This option requires knowledge of functions and calls but is mostly straightforward. Within the subject string, curly braces designate substitution targets and arguments to be inserted—either by *relative* position (`{}`), *absolute* position (e.g., `{1}`), or *keyword* (e.g., `{name}`). As you'll learn when we explore argument passing in depth in [Chapter 18](#), arguments to functions and methods may be passed by position (e.g., *value*) or keyword (e.g., *name=value*), and Python's ability to collect arbitrarily many arguments allows for general method-call patterns. As initial examples:

```
>>> template = '{}', '{}', and {}'                                # Relative position
>>> template.format('expr', 'method', 'fstring')
'expr, method, and fstring'

>>> template = '{0}', '{1}', and {2}'                              # Absolute position
>>> template.format('expr', 'method', 'fstring')
'expr, method, and fstring'

>>> template = '{first}', '{second}', and {third}'                 # Keyword name
>>> template.format(first='expr', second='method', third='fstring')
'expr, method, and fstring'

>>> template = '{first}', {0}, and {third}'                         # Combos ({0} or {})
>>> template.format('method', first='expr', third='fstring')
'expr, method, and fstring'
```

By comparison, the last section's formatting *expression* can be a bit more concise, but uses dictionaries instead of keyword arguments for named references, and as you'll see in a moment doesn't allow quite as much flexibility for value sources in the template string itself (which may be an asset or liability, depending on your perspective):

```
>>> template = '%s, %s, and %s' # Equivalent %s
>>> template % ('expr', 'method', 'fstring')
'expr, method, and fstring'

>>> template = '%(first)s, %(second)s, and %(third)s'
>>> template % dict(first='expr', second='method', third='fstring')
'expr, method, and fstring'
```

Note the use of `dict()` to make a dictionary from keyword arguments here, introduced in [Chapter 4](#) and covered in full in [Chapter 8](#); it's an often less cluttered alternative to the `{...}` literal. Naturally, the subject string in the `format` method call can also be a literal that creates a temporary string, and arbitrary object types can be substituted at targets much like the expression's `%s` code:

```
>>> '{pi}, {} and {years}'.format(62, pi=3.14, years=[1995, 2024])
'3.14, 62 and [1995, 2024]'
```

Just as with the `%` expression and other string methods, `format` creates and returns a new string object, which can be printed immediately or saved for further work (as another reminder, strings are immutable, so `format` really *must* make a new object). String formatting in any of its forms is not just for display:

```
>>> X = '{pi}, {} and {years}'.format(62, pi=3.14, years=[1995, 2024])
>>> X
'3.14, 62 and [1995, 2024]'
```

```
>>> X.split(' and ')
['3.14, 62', '[1995, 2024]']
```

```
>>> Y = X.replace('and', 'but under no circumstances')
>>> Y
'3.14, 62 but under no circumstances [1995, 2024]'
```

Adding keys, attributes, and offsets

Like `%` formatting expressions, `format` calls can become more complex to support more advanced usage. For instance, `format` strings can name object attributes and dictionary keys—as in normal Python syntax, square brackets name dictionary keys and dots denote object attributes of an item referenced by position or keyword. The first of the following examples indexes a dictionary on the key `kind` and then fetches the attribute `platform` from the already imported `sys` module object. The second does the same, but names the objects by keyword instead of position:

```
>>> import sys # Standard-library module

>>> 'This {1[kind]} runs {0.platform}'.format(sys, {'kind': 'laptop'})
'This laptop runs darwin'
```

```
>>> 'This {map[kind]} runs {sys.platform}'.format(sys=sys, map={'kind': 'phone'})
'This phone runs linux'
```

Square brackets in `format` strings can also name offsets to index lists (and other sequences), but only a single positive offset works syntactically within each `[]`, so this feature is not as general as you might think. To reference negative offsets or slices, or to use arbitrary expression results in general, you must run expressions outside the `format` string itself, just as you would for `%` expressions (note the use of `*parts` here to unpack a tuple's items into individual function arguments; you'll learn more about this form when we study function arguments in [Chapter 18](#)):

```
>>> somelist = list('HACK')
>>> somelist
```



```

['H', 'A', 'C', 'K']

>>> 'zero={0[0]}, two={0[2]}'.format(somelist)
'zero=H, two=C'

>>> 'first={}, last={}'.format(somelist[0], somelist[-1])    # [-1] fails in fmt
'first=H, last=K'

>>> parts = (somelist[0], somelist[-1], somelist[1:3])        # [1:3] fails in fmt
>>> 'first={}, last={}, middle={}'.format(*parts)             # Or {0}, {1}, {2}
'first=H, last=K, middle=['A', 'C']"

```

If you simply cannot do without full generality inside format strings, stay tuned for the *f-string* and its arbitrary nested expressions (albeit in a code literal instead of a method object, and at the cost of more redundancy and less utility).

Formatting method custom formats

Another similarity with % expressions is that `format` lets you can achieve more specific layouts with extra format-string syntax. For the formatting method, we use a colon after the possibly empty substitution target's identification, followed by a format specifier that can name the field size, justification, and a specific type code. Here's the formal structure of what can appear as a substitution target in a format string—its four parts are all optional (denoted by surrounding `[]` here, which aren't coded in the format string) and must appear without intervening spaces:

```
[[fieldname][component][!conversionflag][:formatspec]]
```

Text outside a `{}` substitution target is taken literally and may use doubled `{{` and `}}` to escape braces (each is replaced with a single brace). Within a `{}` substitution target:

`fieldname`

Is an optional number or keyword identifying an argument, which may be omitted to reference arguments by relative position

`component`

Is a string of zero or more `.name` or `[index]` (brackets required!) references, which are used to fetch attributes and indexed values of an argument, and may be omitted to use the whole argument value

`conversionflag`

Starts with a `!` if present, which is followed by `s`, `r`, or `a` to call `str`, `repr`, or `ascii` built-in functions on the value, respectively (this may bypass the value's normal formatting)

`formatspec`

Starts with a `:` if present, followed by text that specifies how the value should be presented, including details such as field width, alignment, padding, decimal precision, and so on, and ends with an optional type code

The *formatspec* component after the colon character has a rich format all its own and is formally described as follows. As you'll see later, this part is reused by *f-strings* (again, `[...]` in this denotes an optional component whose square brackets are not coded literally, and spaces aren't allowed between parts but may appear within some):

```
[[fill][align][sign][z][#][0][width][grouping][.precision][typecode]
```

Within this *formatspec* part of the {} substitution target, the salient parts are these:

fill

Can be any fill character other than { or }

align

May be <, >, =, or ^, for left alignment, right alignment, padding after a sign character, or centered alignment

sign

Can be + (to sign all numbers), – (to sign only negatives), or a space (to use a space for positives)

grouping

May be , or _ to request a comma or underscore separator, added for thousands in decimal number type codes, and four-digit groups in nondecimal number type codes (which support only _)

width and precision

Similar to those in the % expression of the preceding section, as shown by examples ahead

typecode

Similar to those in the % expression, with the exceptions described below this list

Others

For numbers, a 0 before *width* enables sign-aware zero-padding (redundantly with some *fill* usage), and # invokes an alternate form (e.g., adding 0b and 0X prefixes for binary and hex type codes b and X).

The *formatspec* may also contain *nested* {} substitution targets for any of its parts, to use argument-list values dynamically (much like the * in formatting expressions). These nested {} use *fieldname* to identify arguments from which values are pulled, and their formatted results are used in place of the nested {}. Nesting may be only one level deep, and *f-strings* (ahead) use an arbitrary expression instead of *fieldname* in a nested {}.

The method's *typecode* options largely overlap with those used in % expressions and listed earlier in [Table 7-4](#), but the formatting method adds a b to display integers in *binary* format (much like using the bin built-in), adds a % to display *percentages*, uses only d for base-10 integers (i and u are unused), uses ! conversion flags for some cases, and requires a string object for s (to flexibly allow any type like the expression's %, either *omit* the type code or *formatspec* in full, or use a !s conversion flag as described earlier).

See Python's library manual for more on substitution syntax that we'll omit here. In addition to the string's format method, a single object may also be formatted with the `format(object, formatspec)` built-in function (which the method uses internally), and may be customized in user-defined classes with the `__format__` operator-overloading method (see [Part VI](#)). Different objects may use different format specifiers, but most follow norms.

Advanced formatting method examples

As you can tell, the syntax in formatting methods can be complex. Because your best ally in such cases is often the interactive prompt, let's turn to some examples. In the following, {0:10} means the first positional argument in a field 10 characters *wide*; {1:<10} means the second positional argument *left-justified* in a 10-character-wide field; ^10 *center aligns* in 10; and {0.platform:>10} means the

platform attribute of the first argument, right-justified in a 10-character-wide field (notice again the use of `dict()` to make a dictionary from keyword arguments):

```
>>> '{0:10} = {1:10}'.format('text', 123.4567)
'text          = 123.4567'

>>> '{0:>10} = {1:<10}'.format('text', 123.4567)
'          text = 123.4567 '

>>> '{1[kind]:^10} = {0.platform:^10}'.format(sys, dict(kind='laptop'))
'    laptop    = darwin    '
```

As demoed earlier, you can *omit* the argument number if you're selecting them from left to right—though this may make your code less explicit, thereby negating one of the purported pluses of `format` versus `%`. Code readers must count to match `{}`s to arguments off to the right (something the f-string's inline expressions wholly avoid):

```
>>> '{:10} = {:10}'.format('text', 123.4567)
'text          = 123.4567'
```

Floating-point numbers support the same *type codes* and formatting specificity in formatting method calls as in `%` expressions. For instance, in the following `{2:g}` means the third argument formatted by default according to the “g” floating-point representation, `{:.2f}` designates the “f” floating-point format with just two decimal digits (and rounding), and `{:06.2f}` denotes a field with a width of six characters and zero padding on the left:

```
>>> '{0:e}, {1:.3e}, {2:g}'.format(3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'

>>> '{:f}, {:.2f}, {:06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Hex, octal, and binary formats are supported by the formatting method as well (`%` has all these *except* binary). In fact, string formatting is an alternative to some of the built-in functions that format integers to a given base:

```
>>> '{:X}, {:o}, {:b}'.format(255, 255, 255)           # Hex, octal, binary
'FF, 377, 11111111'

>>> hex(255), int('FF', 16), 0xFF                     # Other to/from hex
('0xff', 255, 255)

>>> oct(255), int('377', 8), 0o377                     # Other to/from octal
('0o377', 255, 255)

>>> bin(255), int('11111111', 2), 0b11111111          # Other to/from binary
('0b11111111', 255, 255)
```

The formatting method also supports *separator* insertions (but `%` currently does *not*): you can add commas and underscores between thousands groups in numbers, and underscores between four-digit groups in hex, octal, and binary formats, and absolute argument numbers let you *reuse values* passed in (`%` uses dictionaries to do the same):

```
>>> '{:,.2f}'.format(12345.678)
'12,345.68'

>>> '{0:,} {0:_} {1:_x} {1:_b}'.format(2 ** 32, 0x1FFFF)
'4,294,967,296 4_294_967_296 1_ffff 1_1111_1111_1111_1111'
```

Formatting parameters can either be hardcoded in format strings or taken from the arguments list *dynamically* by nested format syntax—much like the `*` syntax in formatting expressions’ width and precision:

```
>>> '{:.4f}'.format(1 / 3.0)           # Parameters hardcoded
'0.3333'
>>> '%.4f' % (1 / 3.0)                 # Ditto for expression
'0.3333'

>>> '{0:.{1}f}'.format(1 / 3.0, 4)     # Take value from arguments
'0.3333'
>>> '%.*f' % (4, 1 / 3.0)              # Ditto for expression
'0.3333'
```

In fact, `format` allows *any* component of a *formatspec* string to be taken from arguments at runtime, rather than hardcoded at programming time (this is more general than `%`). In the following, a number is zero filled, left-justified, signed, twelve wide, comma separated, with two decimal digits—both statically and dynamically (and with and without argument numbering that nearly breaches this page’s width limits!):

```
>>> '{0:0<+12,.2f}!'.format(1234.564)
'+1,234.56000!'

>>> '{0:{1}{2}{3}{4}{5}.{6}{7}}!'.format(1234.564, 0, '<', '+', 12, ',', 2, 'f')
'+1,234.56000!'

>>> '{:{}}{}}{}{}{}{}{}!'.format(1234.564, 0, '<', '+', 12, ',', '.', 2, 'f')
'+1,234.56000!'
```

Finally, Python’s built-in `format` function noted earlier can also be used to format a *single* item. It may be simpler than using the `format` method in this case, and is roughly similar to formatting one item with the `%` expression:

```
>>> '{:.2f}'.format(1.2345)             # String method
'1.23'
>>> format(1.2345, '.2f')               # Built-in function
'1.23'
>>> '%.2f' % 1.2345                     # Expression
'1.23'
>>> f'{1.2345:.2f}'                     # Preview: f-string
'1.23'
```

Technically, the `format` built-in runs the subject object’s `__format__` method, which the `str.format` method does internally for each formatted item. It’s still more verbose than the original `%` expression’s equivalent here, though, and the *f-string* alternative may best both when the value is a variable’s name—which leads us to the next section.

The F-String Formatting Literal

If you’ve survived the formatting story this far, there’s some good news: the third and last variant is mostly just a takeoff on the second. The *f-string* is a text-string literal that *embeds* substitution values in the format string itself, rather than listing them separately. The code it uses to specify custom formatting, though, is the same as that used for the formatting method. Hence, much of what you just learned for the method applies here in full.

F-strings, added in Python 3.6, perform what's generally called *string interpolation*—replacing the text of an expression with the result of running it live, when the f-string itself is run. These expressions are coded inside the f-string and can be arbitrary Python expression code. The effect isn't functionally different from listing replacement values after a % in the expression, or as arguments in the `format` method. Because they embed values where they are to be substituted, though, f-strings are often shorter and may seem easier to read to some observers.

Syntactically, f-strings begin with the letter `f` (uppercase or lowercase, but usually the latter) before any string-literal form (single, double, or triple quotes). The `f` prefix may be combined with `r` in any order to code formatted raw strings that ignore backslashes (e.g., `rf`), and f-strings concatenate implicitly with an adjacent text-string literal of any kind (but use the `f` prefix on other concatenated literals if you want them to be f-strings too—even on continuation lines).

As a preview of [Chapter 37](#), the `f` cannot be combined with byte-string prefix `b` (which means that the f-string, like the `format` method but unlike the % expression, works only for text, not bytes); and cannot be mixed with the backward-compatible and seldom-used `u` (which would yield prefixes inappropriate for this family-oriented text).

F-string formatting basics

Within the f-string literal, curly braces are used to denote substitutions just like the formatting method. Unlike the method, though, `{}`s contain inline Python *expressions* whose formatted runtime results replace the bracketed parts:

```
>>> what = 'coding'
>>> tool = 'Python'

>>> f'Learning {what} in {tool}'
'Learning coding in Python'
```

As usual, the f-string result is a new string that we're letting the REPL display, but it can also be assigned to a name to be used elsewhere in our code. F-strings also frequently appear in `print` calls to display formatted text, though sometimes more often than they should: there's no reason to use an f-string for simple space-separated prints.

In its simplest form like this, an f-string's expressions enclosed in `{}` are evaluated and then formatted per their print-string defaults. This isn't much different from the equivalent expression or method, but is slightly easier on your keyboard and may be slightly easier on your eyes:

```
>>> 'Learning %s in %s' % (what, tool)           # Expression equivalent
'Learning coding in Python'

>>> 'Learning {} in {}'.format(what, tool)       # Method equivalent
'Learning coding in Python'
```

Importantly, *any* expression can be used in the curly braces and works as it would outside the f-string:

```
>>> task = f'Learning {what.upper()} + '!' in {tool + str(3.12)}'
>>> task
'Learning CODING! in Python3.12'
```

Moreover, `{}` expressions are evaluated both *where* they appear (subject to the name-scoping details you'll meet later in this book) and *when* the f-string is run to make a string (not when your code is first read by Python). Like the formatting expression and method, it's a runtime operation that uses the *current* values of any variables it names:

```
>>> what = 'f-strings'
>>> task                                     # F-strings built when run (only)
'Learning CODING! in Python3.12'

>>> task = f'Learning {what.upper()} + '!' in {tool + str(3.12)}'
>>> task
'Learning F-STRINGS! in Python3.12'
```

As a preview, this runtime nature also means that f-strings, unlike all other text-string literals, don't work as [Chapter 15's docstrings](#). This makes sense if you keep in mind that f-strings are runtime code, more like the % expression and format method calls. When coded in a function, for example, an f-string won't be run until that function is called.

One syntax quirk here: as this example demos, you can embed *quotes* within an f-string's {} even if they are the same as the quotes used for the f-string at large—but this is new as of Python 3.12. In earlier Pythons, backslash escapes didn't help for nested quotes, though other enclosing-quote tricks did, and none of this applies outside a {}:

```
f'Learning {what + '!}'          # OK as of Python 3.12

f'Learning {what + '!}'          # An error before Python 3.12
f'Learning {what + '!\\'}'        # And this doesn't make it work

f"Learning {what + '!}"          # OK before (and after) Python 3.12
f'"Learning {what + '!}"'        # Ditto

f'Learning '{what + '!}''        # An error in 3.12+
f'Learning '\\{what + '!}'\\'    # OK if escape quotes outside {}
```

Also new as of Python 3.12, a *backslash* can be used in an f-string's {} part and works just like it does outside the {}, as do both *comments* and *newlines* (the following may stretch the limits of f-strings, but prove these points):

```
>>> f'{'\n'.join([what] * 3) + '\x21'}'
'f-strings\nf-strings\nf-strings!'

>>> f'Learning {                                     # Your comment here
...     what.upper() + '!'
...     } in {tool + str(3.12)}'
'Learning F-STRINGS! in Python3.12'
```

In other words, if you like f-strings, you'll like them best in Python 3.12+. As this book is based on 3.12 (and 3.13 is right around the corner), it will generally use 3.12's f-string rules; mod examples' quotes for older Pythons if needed.

F-string custom formats

As noted, f-strings use the same custom-format syntax as string methods, so there's not much new to learn here. In the abstract, f-strings are coded with a format like this (as usual, [...] means an optional part here and its square brackets are not part of the f-string's code, but spaces are generally allowed between the parts here):

```
f'...literaltext... {expression [=] [!s, !r, or !a] [:formatspec]} ...literaltext...'
```

As in the method, text outside a `{}` is taken literally and uses `{{` and `}}` to escape braces, and the `{}` part can be repeated to embed multiple values. In each, the *expression* part is any Python expression code (including nested f-strings) and is run to produce the substitution value before formatting it. As in the formatting method, the optional `!s`, `!r`, or `!a` render the expression in user-friendly, as-code, or ASCII-with-escapes form—which is the same as calling `str`, `repr`, or `ascii` for the entire expression enclosed by `{}`.

Within a `{}`, the *formatspec* that is coded after a `:` is (nearly) *identical* to that used in the string method and presented earlier, so we won't repeat its syntax here; see “[Formatting method custom formats](#)” on [page 159](#) for the options that f-strings share with the method. As a minor convenience for developers, f-strings also allow an `=` character after *expression* to add the expression's text and an “=” as a label before its value formatted with a `repr` default.

Advanced f-string examples

All of which is easier to explain by example than narrative, so let's get back to running code. Numbers can be formatted in a variety of ways spelled out for the formatting method earlier—including defaults, fixed decimal digits, comma and underscore separators, exponents, signs, and leading zeroes:

```
>>> a = 3.14156
>>> b = 1_234_567

>>> f'{a} and {b}'           # Defaults
'3.14156 and 1234567'

>>> f'{a:.2f} and {b:09}'    # Decimals, padding
'3.14 and 001234567'

>>> f'{a * 1000:,.2f} and {b:,} and {b:_}' # Comma and underscore separators
'3,141.56 and 1,234,567 and 1_234_567'

>>> f'{a * 1000:e} and {b:+012,}' # Exponents, signs, and padding
'3.141560e+03 and +001,234,567'

>>> f'{b:_X} and {b:_o} and {b // 64:_b}' # Hex, octal, binary, underscores
'12_D687 and 455_3207 and 100_1011_0101_1010'
```

Adding an `=` after the expression may be useful when you're debugging code, as it *labels* data automatically (though this may be more readable for simple variable names than larger expressions):

```
>>> f'{a:=e} and {b=:+012,}' # Labeled
'a=3.141560e+00 and b=+001,234,567'

>>> f'{a + 1:=e} and {b * 2=:+012,}'
'a + 1=4.141560e+00 and b * 2=+002,469,134'
```

String formats can vary according to the *s/r/a flag* also coded before the format specifier (but after an `=`). To demo, the following uses a non-ASCII character (an “A” with either an umlaut mark or diaeresis):

```
>>> c = 'h\xc4ck'                                # \xc4 (a.k.a \u00c4) is non-ASCII Ä

>>> f'{c} and {c} and {c}'                        # Defaults
'hÄck and hÄck and hÄck'

>>> f'{c!s} and {c!r} and {c!a}'                  # Display mode, two ways
'hÄck and 'hÄck' and 'h\\xc4ck"'

>>> f'{str(c)} and {repr(c)} and {ascii(c)}'
'hÄck and 'hÄck' and 'h\\xc4ck"'

>>> f'{c=!s} and {c=!r} and {c=!a}'                # Labeled
'c=hÄck and c='hÄck' and c='h\\xc4ck"'

>>> f'{c=!s:8} and {repr(c)} and {c:0>8}'          # Width, fill, alignment
'c=hÄck      and 'hÄck' and 0000hÄck"
```

Advanced tip: as in the formatting method, parts of the format specifier can be fetched *dynamically* at runtime instead of being hardcoded, by using nested {} expressions. The nested expression’s formatted result is used where it appears:

```
>>> width = 8

>>> f'{a:.8f} and {c:0>8}'                        # Hardcoded parameters
'3.14156000 and 0000hÄck'

>>> f'{a:.{width}f} and {c:0>{width}}'            # Dynamic parameters
'3.14156000 and 0000hÄck'

>>> f'{a:.{width}f} and {c * 2:0>{width * 3}}'
'a=3.14156000 and 0000000000000000hÄckhÄck'
```

Like the format method, *any* part of a *formatspec* in a {} can be a nested {}—though they contain *expressions* in the f-string, not argument identifiers. This is why f-string formats are just “(nearly)” identical. Other parts of f-strings don’t allow {}s (forward reference: for economy, the following assigns many names positionally with sequence-assignment syntax covered formally in [Chapter 11](#)—it’s like `a=0`, `b='<'`, and so on):

```
>>> what = 1_234.564
>>> a, b, c, d, e, f, g, h = 0, '<', '+', 12, ',', '.', 2, 'f'

>>> f'{what:0<+12,.2f}!'
'+1,234.56000!'
```

But don't try this at home?

```
>>> f'{what:{a}{b}{c}{d}{e}{f}{g}{h}}!'
'+1,234.56000!'
```

Usage tip: because f-strings run expressions that reference variables, they may not be easy to use as *templates* when substitution values are collected in a *container* at runtime—as they often would be. In the following, a ** converts dictionary keys to keyword arguments, as you’ll learn later in this book:

```
>>> values = dict(tool='Python', role='scripting')    # Collected values

>>> 'Use %(tool)s for %(role)s.' % values             # Expression: keys
'Use Python for scripting.'

>>> 'Use {tool} for {role}.'.format(**values)         # Method: keywords
'Use Python for scripting.'

>>> 'Use {0[tool]} for {0[role]}.'.format(values)     # Method: reused-arg keys
```



```
'Use Python for scripting.'
```

```
>>> f'Use {values['tool']} for {values['role']}.'      # F-string: expressions
'Use Python for scripting.'
```

For more impressive f-string results, assign substitution values to same-scope variables when possible:

```
>>> tool = 'Python'
>>> role = 'scripting'
>>> f'Use {tool} for {role}.'
'Use Python for scripting.'
```

But also bear in mind that f-strings may not be easy to use when templates are loaded from *external files* at runtime—as they often would be. While the format expression and method can treat such templates as simple text data, f-strings are Python *program code*, and hence may have to be run post load with the `eval` function of [Chapter 5](#)—and trusted to not contain code that will do damage (e.g., erasing files is fairly easy in a Python expression):

```
>>> fs = """f'Use {tool} for {role}.'"""              # As if loaded from a file
>>> eval(fs)                                           # Run as code - and trust!
'Use Python for scripting.'
```

F-strings are a powerful tool, but their nested expressions make them geared more toward *in-program* formatting than data-based roles. For more details on the f-string, see its full disclosure in Python's language reference manual.

And the Winner Is...

The previous edition of this book went to considerable lengths (about seven pages) to show how the formatting method, newest at the time, was functionally redundant with the expression, in order to underscore the downsides of feature bloat in programming languages. Given that the number of primary formatting tools in Python has *climbed from two to three* since then, that message may not have entirely hit its mark. Consequently, this edition has dropped most of the rhetoric, and opted to leave this race's call up to you.

But if you're looking for a *guideline*, there is no killer argument for dismissing any formatting option out of hand for all use cases. As stated at the start of this section, *f-strings* may be best in most new code, on logistical grounds alone: newer is less likely to be culled by Python sooner. In fact, we'll be using them regularly in the rest of this book where warranted, so expect more examples ahead. Even so, you will also see the expression and method often in existing code and may *have* to use them in roles that f-strings don't address (e.g., for substitutions in text loaded from files).

More fundamentally, this book is not in the business of telling you what to do, and you are welcome to use *any* formatting option you prefer. Imposing new tools on programmers is exclusive, divisive, and probably rude. Despite norms in the software field today, the choice of development options should be yours—and yours alone—to make.

As for the *bloat*: change is not always bad, but it can be when it creates redundancy or incompatibility. In fact, you've just had a front-row seat to one of its worst consequences: *N* functionally equivalent options can multiply newcomers' learning requirements by *N*. We'll return to this and other perils of ego-fueled churn and convolution in software development at the end of this book. For now, we'll close by simply noting that this stuff still matters. With any luck, a future edition won't have yet another formatting tool to doc, and future learners won't have yet another one to grok.

Plus One More: string.Template

Technically speaking, there are *four* (not three) formatting tools built into Python today, if we include the obscure string module's `Template` tool mentioned earlier. Now that you've seen the other three, you can tell how it compares. The expression, method, and f-string can all be used as templating tools, referring to substitution values by name using dictionary keys, keyword arguments, or variables (the “;” in the following separates multiple statements needed to give the f-string variables to reference):

```
>>> 'The %(num)s %(tool)s' % dict(num=4, tool='formatter')
'The 4 formatters'
>>> 'The {num} {tool}s'.format(num=4, tool='formatter')
'The 4 formatters'
>>> num=4; tool='formatter'; f'The {num} {tool}s'
'The 4 formatters'
```

The module's templating system allows values to be referenced by name too, prefixed by a `$`, as either dictionary keys or keywords, but does not support all the utilities of the other two methods—a limitation that yields simplicity, the prime motivation for this tool:

```
>>> import string
>>> t = string.Template('The $num ${tool}s')
>>> t.substitute(num=4, tool='formatter')
'The 4 formatters'
>>> t.substitute(dict(num=4, tool='formatter'))
'The 4 formatters'
```

See Python's manuals for more details. It's possible that you may see this alternative (as well as additional tools in the third-party domain) in Python code too; thankfully this technique is simple and is used rarely enough to warrant its limited coverage here.

General Type Categories

Now that we've explored the first of Python's collection objects, the string, let's close this chapter by defining a few general type concepts that will apply to most of the types we'll look at from here on. With regard to built-in types, it turns out that operations work the same for all the types in the same category, so we'll only need to define most of these ideas once. We've examined only numbers and strings so far, but because they are representative of two of the three major type categories in Python, you already know more about several other types than you might think.

Types Share Operation Sets by Categories

As you've learned, strings are immutable sequences: they cannot be changed in place (the *immutable* part), and they are positionally ordered collections that are accessed by offset (the *sequence* part). It so happens that all the sequences we'll study in this part of the book respond to the same sequence operations shown in this chapter at work on strings—concatenation, indexing, iteration, and so on. More formally, there are three major type (and hence operation) categories in Python that have this generic nature:

Numbers (integer, floating-point, decimal, fraction, others)

Support addition, multiplication, etc.

Sequences (strings, lists, tuples)

Support indexing, slicing, concatenation, etc.

Mappings (dictionaries)

Support indexing by key, etc.

Python’s byte strings mentioned at the start of this chapter fall under the general “strings” label here; sets are something of a category unto themselves (they don’t map keys to values and are not positionally ordered sequences); and we haven’t yet explored mappings on our in-depth tour (we will in the next chapter). However, many of the other types we will encounter will be similar to numbers and strings. For example, for any sequence objects X and Y :

- $X + Y$ makes a new sequence object with the contents of both operands joined.
- $X * N$ makes a new sequence object with N copies of the sequence operand X .

In other words, these operations work the same way on any kind of sequence, including strings, lists, tuples, and some user-defined object types. The only difference is that the new result object you get back is of the same type as the operands X and Y —if you concatenate lists, you get back a new list, not a string. Indexing, slicing, and other sequence operations work the same on all sequences, too; the type of the objects being processed tells Python which flavor of the task to perform (and if that sounds like *polymorphism* again, it should).

Mutable Types Can Be Changed in Place

The string’s immutable classification is an important constraint to be aware of, yet it tends to trip up new users. If an object type is immutable, you cannot change its value in place; Python raises an error if you try. Instead, you must run code to make a new object containing the new value. The major core types in Python break down as follows:

Immutable (numbers, strings, tuples, frozensets)

None of the object types in the immutable category support in-place changes, though we can always run expressions to make new objects and assign their results to variables as needed.

Mutables (lists, dictionaries, sets, bytearray)

Conversely, the mutable object types can always be changed in place with operations that do not create new objects. Although such objects can be copied manually, in-place changes support direct modification.

Generally, immutable types give some degree of integrity by guaranteeing that an object won’t be changed by another part of a program. For a refresher on why this matters, see the discussion of shared object references in [Chapter 6](#). To see how lists, dictionaries, and tuples participate in type categories, we need to move ahead to the next chapter.

Chapter Summary

In this chapter, we took an in-depth, second-pass tour of the string object type. We learned about coding string literals, and we explored string operations, including sequence expressions, string method calls, and string formatting in its expression, method, and literal flavors. Along the way, we studied a variety of concepts in depth, such as slicing, method call syntax, and triple-quoted block strings. We also defined some core ideas common to a variety of types: sequences, for example, share an entire set of operations demoed here for strings.

In the next chapter, we'll continue our types tour with a look at the most general object collections in Python—lists and dictionaries. As you'll find, much of what you've learned here will apply to those types as well. And as mentioned earlier, in the final part of this book we'll return to Python's string model to flesh out the details of Unicode text and binary data, which are of interest to some, but not all, Python programmers, and depend on tools we haven't yet studied in full. Before moving on, though, here's another chapter quiz to review the material covered here.

Test Your Knowledge: Quiz

1. Can the string `find` method be used to search a list?
2. Can a string slice expression be used on a list?
3. How would you convert a character to its ASCII integer code? How would you convert the other way, from an integer code to a character?
4. How might you go about changing a string in Python?
5. Given a string `S` with the value `'c,od,e'`, name two ways to extract the two characters in the middle.
6. How many characters are there in the string `"a\nb\x1f\000d"`?
7. Write an expression, method call, and f-string to format `'Python'` and `3.12` at a string's beginning and end.

Test Your Knowledge: Answers

1. No, because methods are always type specific; that is, they only work on a single object type. Expressions like `X*Y` and built-in functions like `len(X)` are generic, though, and may work on a variety of types. In this case, for instance, the `in` membership expression has a similar effect as the string `find`, but it can be used to search both strings and lists. Python makes some attempt to name similar methods consistently (many objects have a `copy` method, for example, and mutable objects may share method names like `pop`), but methods are still more type specific than other operation sets.
2. Yes. Unlike methods, expressions are generic and apply to many types. In this case, the slice expression is really a *sequence* operation—it works on any type of sequence object, including strings, lists, and tuples. The only difference is that when you slice a list, you get back a new list.
3. The built-in `ord(S)` function converts from a one-character string to an integer character code; `chr(I)` converts from the integer code back to a string. Keep in mind, though, that these integers are only ASCII codes for text whose characters are drawn only from the ASCII character set. In the Unicode model, text strings are really sequences of Unicode code point identifying integers, which

may fall outside the 7-bit range of numbers reserved by ASCII (we previewed Unicode in [Chapter 4](#) and will revisit it in [Chapter 37](#)).

4. Strings cannot be changed; they are immutable. However, you can achieve a similar effect by creating a new string—by concatenating, slicing, using a method call like `replace`, or running formatting operations—and then assigning the result back to the original variable name.
5. You can slice the string using `S[2:4]` or split on the comma and index the string using `S.split(',')[1]`. Try these interactively to see for yourself.
6. Six. The string `"a\nb\x1f\000d"` contains the characters `a`, newline (`\n`), `b`, literal value 31 (as hex escape `\x1f`, which is a code point that stands for the nonprintable control character US), literal value 0 (an octal escape `\000`), and `d`. Pass the string to the built-in `len` function to verify this and print each of its characters' ord results to see the actual code point (identifying number) values. See [Table 7-2](#) for more details on escapes.
7. There's no right answer for what goes in the middle of the result string, but as examples: `'%s is %s' % ('Python', 3.12)` works for the expression, and `'{} is {}'.format('Python', 3.12)` suffices for the method call. There's almost no reason to use an f-string if all parts are known and formatted per defaults (`f'{'Python'} is {3.12}'` seems silly), but f-strings are more useful if values are first assigned to variables: `x='Python'; y=3.12; f'{x} is {y}'` or similar garners full points.

Lists and Dictionaries

Now that we've explored numbers and strings, this chapter moves on to give the full story on Python's *list* and *dictionary* objects—collections of other objects, and the main workhorses in almost all Python scripts. As you'll find, both are remarkably flexible: they can be changed in place, can grow and shrink on demand, and may contain and be nested in any other kind of object. By leveraging these built-in object types, you can create and process rich information structures in your scripts without having to define new object types of your own.

Lists

The first stop on this chapter's tour is the Python *list*. Lists are Python's most flexible ordered collection object type. Unlike strings, lists can contain any sort of object: numbers, strings, and even other lists. Also, unlike strings, lists may be changed in place by assignment to offsets and slices, list method calls, deletion statements, and more—they are *mutable* objects.

Python lists do the work of many of the collection data structures you might have to implement manually in lower-level languages such as C. Here is a quick look at their main properties. Python lists are:

Ordered collections of arbitrary objects

From a functional view, lists are just places to collect other objects so you can treat them as groups. Lists also maintain a left-to-right positional ordering among the items they contain.

Accessed by offset

Just as with strings, you can fetch a component object from a list by indexing the list on the object's offset. Because items in lists are ordered by their positions, you can also do tasks such as slicing and concatenation.

Variable-length, heterogeneous, and arbitrarily nestable

Unlike strings, lists can grow and shrink in place (their lengths can vary), and they can contain any sort of object, not just one-character strings (they're heterogeneous). Because lists can both contain and be contained by other collection objects, they also support arbitrary nesting; you can create lists of lists of lists, and so on.

Of the category “mutable sequence”

In terms of our type category qualifiers, lists are mutable (i.e., can be changed in place) and can respond to all the *sequence* operations used with strings, such as indexing, slicing, and concatenation. In fact, sequence operations work the same on lists as they do on strings; the only difference is that sequence operations such as concatenation and slicing return new *lists* instead of new strings when applied to lists. Because lists are *mutable*, however, they also support other operations that strings don’t, such as deletion, expansion, and index assignment operations, which change the lists in place.

Arrays of object references

Technically, Python lists contain zero or more *references* to other objects. Lists might remind you of arrays of pointers (addresses) if you have a background in some other languages, and fetching an item from a Python list is about as fast as indexing a C array. In fact, lists really *are* arrays inside the standard CPython interpreter, not linked structures. As we learned in [Chapter 6](#), though, Python always follows a reference to an object whenever the reference is used, so your program deals only with objects. Whenever you assign an object to a data structure component or variable name, Python always stores a reference to that same object, not a copy of it (though the object stored may be a copy of another, if you requested one before the store).

As a preview and reference, [Table 8-1](#) summarizes common and representative list object operations. It is fairly complete, but for the full story, consult the Python standard-library manual, or run a `help(list)` or `dir(list)` call interactively for a complete list of list methods—you can pass in a real list, or the word `list`, which is the name of the list data type. The set of methods here is especially prone to change, so be sure to cross-check in the future.

Table 8-1. Common list literals and operations

Operation	Interpretation
<code>L = []</code>	An empty list
<code>L = [123, 'abc', 1.23, {}]</code>	Four items: indexes 0..3
<code>L = ['Pat', 40.0, ['dev', 'mgr']]</code>	Nested sublists
<code>L = list('code')</code> <code>L = list(range(-4, 4))</code>	List of an iterable’s items, list of successive integers
<code>L[i]</code> <code>L[i][j]</code> <code>L[i:j]</code> <code>len(L)</code>	Index, index of index, slice, length
<code>L1 + L2</code> <code>L * 3</code>	Concatenate, repeat
<code>L1 > L1</code> , <code>L1 == L2</code>	Comparisons: magnitude, equality
<code>3 in L</code> <code>for x in L: print(x)</code>	Membership, iteration
<code>L.append(4)</code> <code>L.extend([5, 6, 7])</code> <code>L.insert(i, X)</code>	Methods: growing
<code>L.index(X)</code> <code>L.count(X)</code>	Methods: searching

Operation	Interpretation
L.sort() L.reverse() L.copy() L.clear()	Methods: sorting, reversing, copying, clearing
L.pop(i) L.remove(X) del L[i] del L[i:j] L[i:j] = []	Methods, statements: shrinking
L[i] = 3 L[i:j] = [4, 5, 6]	Index assignment, slice assignment
L = [*x, 0, *y, *z]	Iterable unpacking
L = [x**2 for x in range(5)] list(map(ord, 'python'))	List comprehensions and maps

When written down as a *literal* expression, a list is coded as a series of objects (really, expressions that return objects) in square brackets, separated by commas. For instance, the second row in [Table 8-1](#) assigns the variable `L` to a four-item list. A *nested* list is coded as a nested square-bracketed series (row 3), and the *empty* list is just a square-bracket pair with nothing inside (row 1).¹

Many of the operations in [Table 8-1](#) should look familiar, as they are the same sequence operations we put to work on strings earlier—indexing, concatenation, iteration, and so on. Lists also respond to list-specific method calls (which provide utilities such as sorting, reversing, adding items to the end, etc.), as well as in-place change operations (deleting items, assignment to indexes and slices, and so forth). Again, lists have these tools for change operations because they are a mutable object type.

Lists in Action

Probably the best way to understand lists is to see them at work. Let’s once again turn to some simple interpreter interactions to illustrate the operations in [Table 8-1](#).

Basic List Operations

Because they are *sequences*, lists support many of the same operations as strings, which means we don’t have to repeat all the operation details again here. In short, though, lists respond to the `+` and `*` operators much like strings—they mean concatenation and repetition here too, except that the result is a new list, not a string:

```
$ python3
>>> len([1, 2, 3])
3
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> ['Py!'] * 4
['Py!', 'Py!', 'Py!', 'Py!']
```

Launch a REPL
Length
Concatenation
Repetition

¹ In practice, you won’t see many lists written out like this in list-processing programs. It’s more common to see code that processes lists constructed dynamically (at runtime), from user inputs, file contents, and so on. In fact, although it’s important to master literal syntax, many data structures in Python are built by running program code at runtime.

Although the `+` operator works the same for lists and strings, it's important to know that it expects the *same* sort of sequence on both sides—otherwise, you get a type error when the code runs. For instance, you cannot concatenate a list and a string unless you first convert the list to a string (using tools such as `str` or formatting) or convert the string to a list (the `list` built-in function does the trick):

```
>>> str([1, 2]) + '34'                # Same as '[1, 2]' + '34'
'[1, 2]34'
>>> [1, 2] + list('34')               # Same as [1, 2] + ['3', '4']
[1, 2, '3', '4']
```

As suggested in prior chapters, lists also support *comparisons*, which automatically compare all parts from left to right until a result is known. In the following, for instance, a nested 3 is greater than a nested 2, and the one-item list `[1]` at the end is considered less because it's shorter (though it broadly prefers the term horizontally challenged):

```
>>> L = [1, [2, 3], 4]
>>> (L == [1, [2, 3], 4]), (L > [1, [2, 2], 4]), (L > [1])
(True, True, True)
```

Indexing and Slicing

Because lists are sequences, indexing and slicing also work the same way for lists as they do for strings. For lists, though, the result of indexing is whatever type of object lives at the offset you specify (not a one-character string), and slicing a list always returns a new list (not a string):

```
>>> L = ['hack', 'Hack', 'HACK!']
>>> L[2]                                # Offsets start at zero
'HACK!'
>>> L[-2]                              # Negative: count from the right
'Hack'
>>> L[1:]                               # Slicing fetches sections
['Hack', 'HACK!']
```

New here: because you can nest lists and other object types within lists, you will sometimes need to string together index operations to go *deeper* into a data structure. For example, one of the simplest ways to represent matrixes (multidimensional arrays) in Python is as lists with nested sublists. Here's a basic 3×3 two-dimensional list-based array—a reprise from [Chapter 4](#):

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

With one index, you get an entire row (really, a nested sublist), and with two, you get an item within the row:

```
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
```

As demoed earlier, lists can naturally span multiple lines if you want them to because they are contained by a pair of brackets; watch for more on syntax like this in the next part of the book (and ignore the “...” if absent in your REPL):

```
>>> matrix = [[1, 2, 3],
...           [4, 5, 6],
...           [7, 8, 9]]
>>> matrix[1][1]
5
```

For more on matrixes, also watch for a dictionary-based alternative later in this chapter, which can be more efficient when matrixes are largely empty. We'll also continue this thread in [Chapter 20](#) where we'll write additional matrix code, especially with list comprehensions. And for high-powered numeric work, the *NumPy* extension mentioned in [Chapters 1, 4, and 5](#) provides other ways to handle matrixes.

Changing Lists in Place

Because lists are mutable, they support operations that change a list object *in place*. That is, list operations in this section and others all modify the list object directly—overwriting its former value—without requiring that you make a new copy, as you had to for strings. Because Python deals only in object references, this distinction between changing an object in place and creating a new object matters; as discussed in [Chapter 6](#), if you change an object in place, you might impact more than one reference to it at the same time.

Index and slice assignments

First up in this category is a twist on the indexing and slicing we've already explored. When using a list, you can change its contents by *assigning* to either a particular item (*offset*) or an entire section (*slice*):

```
>>> L = ['code', 'Code', 'CODE!']
>>> L[1] = 'Hack'           # Index assignment
>>> L                       # Replaces item 1
['code', 'Hack', 'CODE!']

>>> L[0:2] = ['write', 'Python'] # Slice assignment: "delete+insert"
>>> L                       # Replaces items 0,1
['write', 'Python', 'CODE!']
```

Both index and slice assignments are in-place changes—they modify the subject list directly, rather than generating a new list object for the result. *Index assignment* in Python works much as it does in most other languages: Python replaces the single object reference at the designated offset with a new one; the offset's reference is changed.

Slice assignment, the last operation in the preceding example, replaces an entire section of a list in a single step. Because it can be a bit complex, it is perhaps best thought of as a combination of two steps:

1. *Deletion*: The slice you specify to the left of the = is deleted.
2. *Insertion*: The new items contained in the iterable object to the right of the = are inserted into the list on the left, at the place where the old slice was deleted.²

This isn't what really happens, but it can help clarify why the number of items inserted doesn't have to match the number of items deleted. For instance, given a list *L* of two or more items, an assignment *L*[1:2]=[4,5] replaces one item with two—it's as though Python first deletes the one-item slice at

2 This description requires elaboration when the value and the slice being assigned *overlap*: *L*[2:5]=*L*[3:6], for instance, works fine because the value to be inserted is fetched *before* the deletion happens on the left. Hence, slice assignment is really a *fetch + delete + insert*, but the fetch part matters too rarely to make the marquee.

[1:2] (from offset 1, up to but not including offset 2), then inserts both 4 and 5 where the deleted slice used to be. The net effect makes the list *larger*.

This also explains why the second slice assignment in the following is really an *insert*—Python replaces an empty slice at [1:1] with two items; and why the third is really a *deletion*—Python deletes the slice (the item at offset 1), and then inserts nothing:

```
>>> L = [1, 2, 3]
>>> L[1:2] = [4, 5]           # Replacement/insertion
>>> L
[1, 4, 5, 3]
>>> L[1:1] = [6, 7]          # Insertion (replace nothing)
>>> L
[1, 6, 7, 4, 5, 3]
>>> L[1:2] = []              # Deletion (insert nothing)
>>> L
[1, 7, 4, 5, 3]
```

In effect, slice assignment replaces an entire section, or “column,” all at once—even if the column or its replacement is empty. Because the length of the sequence being assigned does not have to match the length of the slice being assigned to, slice assignment can be used to *replace* (by overwriting), *expand* (by inserting), or *shrink* (by deleting) the subject list. It’s a powerful operation, but frankly, one that you may not see very often in practice. There are often more straightforward and mnemonic ways to replace, insert, and delete (including concatenation expressions, and the `insert`, `pop`, and `remove` list methods coming up soon), which Python programmers tend to prefer in practice.

On the other hand, this operation can be used as a sort of in-place concatenation at the front of the list—per the next section’s method coverage, something the list’s `extend` does more mnemonically but at list end:

```
>>> L = [1]
>>> L[:0] = [2, 3, 4]         # Insert all at :0, an empty slice at front
>>> L
[2, 3, 4, 1]
>>> L[len(L):] = [5, 6, 7]    # Insert all at len(L):, an empty slice at end
>>> L
[2, 3, 4, 1, 5, 6, 7]
>>> L.extend([8, 9, 10])      # Insert all at end, named method (see also +=)
>>> L
[2, 3, 4, 1, 5, 6, 7, 8, 9, 10]
```

List method calls

Like strings, Python list objects also support type-specific method calls, most of which change the subject list itself:

```
>>> L = ['write']
>>> L.append('Python')        # Append: add one item at the end
>>> L
['write', 'Python']
>>> L.extend(['code', 'goodly']) # Extend: add many items at the end
>>> L
['write', 'Python', 'code', 'goodly']
>>> L.sort()                  # Sort: order list items ('P' < 'c')
>>> L
['Python', 'code', 'goodly', 'write']
```

Methods were introduced in [Chapter 7](#). In brief, they are functions (really, object attributes that reference functions) that are associated with and act upon particular subject objects (the objects through

which they are called). Methods provide type-specific tools; the list methods presented here, for instance, are generally available only for lists.

Demoed in the preceding code, `append` is perhaps the most commonly used list method. It simply tacks a *single* item (really, an object reference) onto the end of the subject list *in place*, and the list expands to make room for the addition. Unlike concatenation, `append` expects you to pass in one object, not a list, and uses it literally. In fact, the effect of `L.append(X)` is similar to `L+[X]`, but the former changes `L` in place, while the latter makes a new list.³

By contrast, the `extend` method adds *multiple* items at the end of the list, again in place. Technically, `extend` always iterates through and adds each item in the passed *iterable* object, whereas `append` simply adds a single item as is without iterating—a distinction that will be more meaningful in [Chapter 14](#). For now, it's enough to know that `extend` adds many items, and `append` adds one. Also ahead: [Chapter 11](#) covers the `+=` statement, which does in-place assignment for lists too, and is yet another way to add many items that redundantly mirrors `extend`.

The `sort` method *orders* the list's items but merits a section of its own.

Sorting lists

As we've just witnessed, `sort` orders a list in place. It's a common tool that uses Python standard comparison tests (string comparisons in most examples here, but other objects work in sorts too), and by default sorts in ascending order. You can modify sort behavior by passing in *keyword arguments*—a special *name=value* syntax in function calls that we've used in earlier chapters, gives values by name, and is often used for configuration options.

In sorts, the `reverse` argument allows sorts to be made in descending instead of ascending order, and the `key` argument gives a one-argument function that returns the value to be used in sorting—the string object's standard lower case converter in the following (though its newer casefold may handle some types of Unicode text better):

```
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort()                                     # Sort with mixed case
>>> L
['ABD', 'aBe', 'abc']

>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower)                       # Normalize to lowercase
>>> L
['abc', 'ABD', 'aBe']

>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower, reverse=True)        # And change sort order
>>> L
['aBe', 'ABD', 'abc']
```

³ Unlike `+` concatenation, `append` doesn't have to generate new objects, so it's usually faster than `+` too. You can also mimic `append` with the clever slice assignments of the prior section: `L[len(L):]=[X]` is like `L.append(X)`, and `L[:0]=[X]` is like appending at the front of a list. Both delete an empty slice and insert `X`, changing `L` in place quickly, like `append`. Both are arguably more complex than list methods, though. For instance, `L.insert(0, X)` can also append an item to the front of a list, and seems noticeably more mnemonic. `L.insert(len(L), X)` inserts one object at the end too, but unless you like typing, you might as well use `L.append(X)`!

The sort key argument can also be useful when sorting lists of dictionaries, to select a sort value by indexing each dictionary on a field along the way. We'll study dictionaries later in this chapter, and you'll learn more about keyword function arguments in [Part IV](#).

Two notes of caution here: first, sorts won't work by default with *mixed types*. In Python, magnitude comparison of mixed types is an error, as it's generally ambiguous. Because sorting uses these comparisons internally, though, sorting mixed-type lists fails by proxy. To work around this limitation, use the key argument to code value transformations during the sort. The following simply converts all items to strings with the `str` built-in, but key can be an arbitrary function of your making (and is often coded inline with the `lambda` expression you'll meet later in this book):

```
>>> L = [1, 'hack', 2]          # Mixed-type sorts fail by default
>>> L.sort()
TypeError: '<' not supported between instances of 'str' and 'int'

>>> L.sort(key=str)
>>> L                            # Enable mixed-type sorts: all str
[1, 2, 'hack']
```

Second, beware that `append`, `extend`, and `sort` change the associated list object *in place*, but don't return the modified list as a result (technically, they return the `None` placeholder object introduced in [Chapter 4](#)). If you run `L=L.append(X)`, you won't get the modified value of `L`; in fact, you'll lose the reference to the list altogether! When you use methods like these, objects are changed as a side effect, so there's no reason to reassign.

Partly because of such constraints, sorting is also available in Python as the `sorted` built-in *function*, which sorts *any* collection (not just lists) and returns a *new* list for the result (instead of changing the collection in place):

```
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted(L)                    # Sorting built-in function
['ABD', 'aBe', 'abc']
>>> L                            # L is not modified
['abc', 'ABD', 'aBe']

>>> sorted(L, key=str.lower, reverse=True)    # Same arguments as list.sort
['aBe', 'ABD', 'abc']

>>> L = ['abc', 'ABD', 'aBe']
>>> sorted([x.lower() for x in L], reverse=True) # Pretransform items: differs!
['abe', 'abd', 'abc']
```

Notice the last example here—we can convert to lowercase prior to the sort with a list comprehension (introduced in [Chapter 4](#) and expanded shortly), but the result does not contain the *original* list's values as it does with the key argument. The latter is applied temporarily during the sort, instead of changing the values to be sorted altogether. As we move along, you'll see roles in which the `sorted` built-in can sometimes be more useful than the `sort` method.

More List Methods

Like strings, lists have other methods that perform other specialized operations. For instance, `reverse` reverses the list in place, and `pop` deletes one item at the end (by default). Just like sorting, there is also a reversed built-in function that does not change the list in place. Confusingly, though, `reversed` does not return a new list like `sorted`; it instead returns an iterable object that produces results on demand,

and must be wrapped in a `list` call to collect its results if a real list is needed (e.g., for indexing, or display at the REPL here):

```
>>> L = [1, 2, 3, 4, 5]
>>> L
[1, 2, 3, 4, 5]
>>> L.pop()                                # Delete and return an item: @ -1 by default
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()                            # In-place reversal
>>> L
[4, 3, 2, 1]
>>> list(reversed(L))                      # Reversal built-in with a result (iterable)
[1, 2, 3, 4]
>>> L                                      # L was unchanged
[4, 3, 2, 1]
```

In some types of programs, the list `pop` method is used in conjunction with `append` to implement a quick last-in-first-out (LIFO) *stack* structure. The end of the list serves as the “top” of the stack:

```
>>> L = []
>>> L.append(1)                            # Push onto stack
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop()                                # Pop off stack
2
>>> L
[1]
```

The `pop` method also accepts an optional offset of the item to be deleted and returned (the default is the last item at offset `-1`). Other list methods remove an item by value (`remove`), insert an item at an offset (`insert`), count the number of occurrences (`count`), and search for an item’s offset (`index`—a search for the *index* of an item, not to be confused with indexing itself, despite the name!):

```
>>> L = ['hack', 'Py', 'code']
>>> L.index('Py')                          # Index_of_ an object (search/find)
1
>>> L.insert(1, 'more')                   # Insert at offset/position
>>> L
['hack', 'more', 'Py', 'code']
>>> L.remove('code')                      # Delete by value
>>> L
['hack', 'more', 'Py']
>>> L.pop(1)                             # Delete by offset/position
'more'
>>> L
['hack', 'Py']
>>> L.count('Py')                         # Number of occurrences
1
```

Note that unlike other list methods, `count` and `index` do not change the list itself, but return information about its content. Run a `help(list)` or `dir(list)` in your REPL, see Python reference resources, or experiment with these calls interactively on your own to learn more about list methods.

Iteration, Comprehensions, and Unpacking

Lists also respond to other sequence operations we used on strings in the prior chapter, including *iteration* tools, and are regularly used in conjunction with the `range` built-in previewed in [Chapter 4](#):

```
>>> 3 in [1, 2, 3]                # Membership
True
>>> for x in [1, 2, 3]:
...     print(x, end=' ')          # Iteration
...
1 2 3
>>> list(range(5))                # Counter generators (0..N-1)
[0, 1, 2, 3, 4]
```

We will talk more formally about for iteration and the `range` built-in in [Chapter 13](#), because they are related to statement syntax. Per [Chapter 4](#) previews, though, for loops step through items in any sequence (or other iterable) from left to right, executing one or more statements for each item; and `range` makes a series of integers, for use in a variety of roles, including counter loops (and coerced by `list` to surrender its values for display here).

List comprehensions and maps

The last items in [Table 8-1](#), list comprehensions and `map` calls, were also introduced in [Chapter 4](#) and are covered in full in [Chapters 14](#) and [20](#). Their basic usage is straightforward, though—list *comprehensions* are close kin to for loops, and build a new list by applying an expression to each item in a sequence (really, any iterable):

```
>>> res = [x * 4 for x in 'code']    # List comprehension
>>> res
['cccc', 'oooo', 'dddd', 'eeee']
```

As a preview, this expression is equivalent to a for loop that builds up a list of results manually with the `append` we met earlier, but as you'll learn in later chapters, list comprehensions are simpler to code and may run faster:

```
>>> res = []
>>> for x in 'code':
...     res.append(x * 4)
...
>>> res
['cccc', 'oooo', 'dddd', 'eeee']
```

Comprehension also support extensions like `if` filters and nested for loops, and the `map` built-in function does similar work, but applies a *function* instead of an expression to items in a sequence (or other iterable) and collects all the results in a new list—if we wrap it in `list` to force it to produce all its results:

```
>>> [x * 4 for x in 'program' if x >= 'p']    # Filter items with if clauses
['pppp', 'rrrr', 'rrrr']

>>> [x + y for x in 'py' for y in '312']      # Nested loops: see Chapter 14!
['p3', 'p1', 'p2', 'y3', 'y1', 'y2']

>>> list(map(abs, [-1, -2, 0, 1, 2]))         # Map a function across a sequence
[1, 2, 0, 1, 2]
```


While these are powerful tools, their loop equivalents also make them optional. Stay tuned for the related dictionary comprehension later in this chapter, and much more on comprehensions and maps later in this book.

List-literal unpacking

As of Python 3.5, list literals also support a `*` syntax that *unpacks* the contents of any iterable (including sequences like lists and strings) at the top level of the list being created. The effect flattens the starred item in the new list:

```
>>> L = ['code', 'hack']
>>> [L, 2, 3, L]
[['code', 'hack'], 2, 3, ['code', 'hack']] # Normal assignments nest items

>>> [*L, 2, 3, *L] # Iterable unpacking in list literal
['code', 'hack', 2, 3, 'code', 'hack']

>>> S = 'code' # Any sequence (or other iterable)
>>> [*Py, *S, *range(3)]
['P', 'y', 'c', 'o', 'd', 'e', 0, 1, 2]
```

Importantly, and as is so often true in Python today, `*` unpacking is never required. The following simple *concatenation*, for example, has the same effect as the preceding's first `*` unpacking—which is redundant in this case:

```
>>> L + [2, 3] + L # What * unpacking does, for lists
['code', 'hack', 2, 3, 'code', 'hack']
```

For nonlist iterables, unpacking without `*` requires *conversions* because `+` expects lists on both sides, but it's not much extra work, especially given the rarity of such code:

```
>>> list('Py') + list(S) + list(range(3)) # What * unpacking does, for nonlists
['P', 'y', 'c', 'o', 'd', 'e', 0, 1, 2]
```

Loops can achieve the same goals as unpacking too; they may be more verbose, but are also more general purpose:

```
>>> M = []
>>> for x in ('Py', S, range(3)): M.extend(x) # Loops v special-case hacks
...
>>> M
['P', 'y', 'c', 'o', 'd', 'e', 0, 1, 2]
```

Notice two things about this example. First, it codes a *tuple* of three items as the sequence that the for loop steps through (the tuple's outer parentheses are optional but explicit). Second, the for loop in this code avoids a line by moving its body up to its *header*, using a syntax rule we'll define formally in the next part of this book (tl;dr: this works only for simple statements, like calls to `print` and `extend`).

Because we're not quite ready for the full iteration story, we'll postpone further details for now, but watch for similar unpacking expressions for dictionaries later in this chapter. Here, though, you should already be able to weigh for yourself whether the alternatives to `*` unpacking were sufficiently subpar to warrant the convolution.

Other List Operations

Because lists are mutable, you can also use the `del` statement to delete an item or section in place:

```
>>> L = ['hack', 'more', 'Py', 'code']
>>> del L[0]                                # Delete one item (in place)
>>> L
['more', 'Py', 'code']
>>> del L[1:]                               # Delete an entire section
>>> L                                       # Same as L[1:] = []
['more']
```

As covered earlier, because *slice assignment* is a deletion plus an insertion, you can also delete a section of a list by assigning an empty list to a slice (`L[i:j]=[]`); Python deletes the slice named on the left, and then inserts nothing. Assigning an empty list to an index, on the other hand, just stores a reference to the empty list object in the specified slot, rather than deleting an item:

```
>>> L = ['hack', 'more', 'Py', 'code']
>>> L[0:1] = []
>>> L
['more', 'Py', 'code']
>>> L[1:] = []
>>> L
['more']
>>> L[0] = []
>>> L
[[]]
```

Although all the operations just discussed are typical, there may be additional list methods and operations not illustrated here. The method toolbox may also change over time, and in fact has: the newer `L.copy()` method makes a top-level copy of the list, much like `L[:]` and `list(L)`, but is symmetric with copy methods in sets and dictionaries. For a comprehensive and up-to-date list of type tools, you should always consult Python's manuals.

And because it's such a common hurdle, this book is compelled to remind you once again that all the in-place change operations discussed here work only for *mutable* objects: they won't work on strings (or tuples, coming up in [Chapter 9](#)), no matter how hard you try. Mutability is an inherent yes/no property of each object type—including the subject of this chapter's next section.

Dictionaries

Along with lists, *dictionaries* are one of the most flexible built-in object types in Python. If you think of lists as order-based collections of objects, you can think of dictionaries as key-based collections; the chief distinction is that in dictionaries, items are stored and fetched by *key*, instead of by positional offset. While lists can serve roles similar to arrays in other languages, dictionaries can take the place of records, search tables, and any other sort of aggregation where item names are more meaningful than item positions.

For example, dictionaries can replace many of the searching algorithms and data structures you might have to implement manually in lower-level languages—as a highly optimized built-in tool, indexing a dictionary is a very fast search operation. Dictionaries also sometimes do the work of records and symbol tables used in other languages, can be used to represent sparse (mostly empty) data structures, and much more. As a rundown of their main properties, Python dictionaries are:

Accessed by key, not offset position

Dictionaries are sometimes called *associative arrays* or *hashes* (especially by users of other scripting languages). They associate a set of values with corresponding keys, so you can fetch an item out of a dictionary using the key under which you originally stored it. You use the same indexing operation to get components in a dictionary as you do in a list, but the index takes the form of a key, not a relative offset.

Insertion-ordered collections of arbitrary objects

Unlike in a list, keys in a dictionary are ordered only by the order in which they are *inserted*. This is not the same as the positional ordering of sequences like lists: items added to dictionaries later always show up at the end of the keys list (even if they appeared earlier in the past), and there is no way to insert a key into the middle of the keys list. Moreover, this ordering is new as of Python 3.7, before which keys' left-to-right order was pseudo random. Under either ordering regime, keys provide the symbolic (not physical) locations of items in a dictionary.

Variable-length, heterogeneous, and arbitrarily nestable

Like lists, dictionaries can grow and shrink in place (without new copies being made), they can contain objects of any type, and they support nesting to any depth (they can be freely mixed with lists, other dictionaries, and so on). Each key can have just one associated value, but that value can be a *collection* of multiple objects if needed, and a given value can be stored under any number of keys.

Of the category “mutable mapping”

You can change dictionaries in place by assigning to indexes (they are *mutable*), but they don't support the sequence operations that work on strings and lists. Because dictionaries are key-based collections, operations that depend on a fixed positional order (e.g., concatenation, slicing) don't make sense. Instead, dictionaries are the only built-in, core-type representatives of the *mapping* category—objects that map keys to values. Other mappings in Python are created by imported modules, not language syntax.

Tables of object references (hash tables)

If lists are arrays of object references that support access by position, dictionaries are tables of object references that support access by key. Internally, dictionaries are implemented as hash tables (data structures that support very fast retrieval), which start small and grow on demand. Moreover, Python employs optimized hashing algorithms to find keys, so retrieval is quick. Like lists, though, dictionaries store object *references* (not copies, unless you make them explicitly before storing).

For reference and preview, [Table 8-2](#) summarizes some of the most common and representative dictionary operations and is relatively complete at this writing. As usual, though, you should consult Python's library manual or run a `dir(dict)` or `help(dict)` call for a complete list (`dict` is the built-in name of the dictionary type).

Table 8-2. Common dictionary literals and operations

Operation	Interpretation
<code>D = {}</code>	Empty dictionary
<code>D = {'name': 'Pat', 'age': 40.0}</code>	Two-item dictionary
<code>E = {'cto': {'name': 'Sue', 'age': 40}}</code>	Nesting

Operation	Interpretation
D = dict(name='Bob', age=40) D = dict(['name', 'Bob'], ('age', 40)) D = dict(zip(keylist, valueslist)) D = dict.fromkeys(['name', 'age'])	Alternative construction techniques: keywords, key/value pairs, zipped key/value lists, key lists
D['name'] E['cto']['age']	Indexing by key, nested indexes
'age' in D for key in D: print(D[key])	Key membership and iteration
D.keys() D.values() D.items() D.copy() D.clear() D.update(D2) D.get(key, default?) D.pop(key, default?) D.setdefault(key, default?) D.popitem()	Methods: all keys, all values, all key+value tuples, copy (top-level), clear (remove all items), merge by keys, fetch by key, if absent default (or None), remove by key, if absent default (or error) fetch by key, if absent set default (or None), remove/return any (key, value) pair; etc.
len(D)	Length: number of stored entries
D[key] = 62	Adding keys, changing key values
del D[key]	Deleting entries by key
D1 == D2	Comparisons: equality (only)
list(D.keys()) D1.keys() & D2.keys()	Dictionary views
D = {**x, 'a': 1, **y, **z}	Dictionary unpacking
D = {x: x*2 for x in range(10)}	Dictionary comprehensions
D E	Dictionary merge expression: copy + update

Per [Table 8-2](#), when coded as a *literal* expression, a dictionary is written as a series of *key:value* pairs separated by commas, and enclosed in curly braces.⁴ An empty dictionary is an empty set of curly braces, and you can nest dictionaries by simply coding one as a value inside another dictionary, or within a list or tuple literal. There’s a lot more to dictionaries than their literals, though, as the next section’s tutorial will begin to reveal.

Dictionaries in Action

As [Table 8-2](#) summarizes, dictionaries are indexed by key, and nested dictionary entries are referenced by a series of indexes (keys in square brackets). When Python creates a dictionary, it stores its items in a way that associates values with keys; to fetch a value back, you supply the key with which it is associated, not its relative position. Let’s go back to the interactive prompt to see what this all looks like in code.

⁴ As for lists, you might not see dictionaries coded in full using literals very often—programs rarely know all their data before they are run, and more typically extract it dynamically from users, files, and so on. Lists and dictionaries are grown in different ways, though. In the next section you’ll see that you often build up dictionaries by assigning to new keys at runtime; this approach fails for lists, which are commonly grown with `append` or `extend` instead.

Basic Dictionary Operations

In normal operation, you create dictionaries with literals and store and access items by key with indexing:

```
$ python3
>>> D = {'hack': 1, 'Py': 2, 'code': 3}      # Make a dictionary
>>> D['Py']                                   # Fetch a value by key
2
>>> D                                         # Insertion ordered
{'hack': 1, 'Py': 2, 'code': 3}
```

Here, the dictionary is assigned to the variable `D`; the value of the key `'Py'` is the integer 2, and so on. We use the same square bracket syntax to index dictionaries by key as we did to index lists by offset, but here it means access by key, not by position. Notice the end of this example: unlike the randomly ordered sets we studied in [Chapter 5](#), dictionary keys retain their insertion order today, but we need to cover some more basics before formalizing this.

The built-in `len` function works on dictionaries, too—it returns the number of values stored in the dictionary or, equivalently, the number of its keys. The dictionary `in` membership operator allows you to test for key existence, and the `keys` method returns all the keys in the dictionary. The latter of these can be useful for processing dictionaries in full, by fetching corresponding values in loops. Because the keys result can be treated like a normal list, it can also be sorted if order matters and the automatic insertion order doesn't do the job (more on sorting and dictionaries later):

```
>>> len(D)                                   # Number of entries in dictionary
3
>>> 'code' in D                               # Key membership test
True
>>> list(D.keys())                           # Create a new list of D's keys
['hack', 'Py', 'code']
```

Notice the second step in this listing. As noted, the `in` membership test used for strings and lists also works on dictionaries, where it checks whether a key is stored. Technically, this works because dictionaries define a protocol run by `in` to lookup a key quickly. Other objects provide `in` protocols that reflect their common uses; files, for example, use iterators that read line by line. This will matter later in this book, when we reach iterators and classes.

Also note the syntax of the last step in this listing. It uses `list` for similar reasons—the dictionary `keys` method returns an iterable “view” object that produces results on demand, instead of a physical list. The `list` call forces it to serve up all its values at once for display at the interactive REPL, though this call isn't required and may even be subpar in some other contexts. More on this, as well as other dictionary basics like comparisons, later in this chapter.

Changing Dictionaries in Place

Let's continue with our interactive session. Dictionaries, like lists, are *mutable*, so you can change, expand, and shrink them in place without making new dictionaries: simply assign a value to a key to change or create an entry. The `del` statement works here, too; it deletes the entry associated with the key specified as an index. Notice also the nesting of a list inside a dictionary in this example (the value of the key `'Py'`); all collection data types in Python can nest inside each other arbitrarily, and can be changed independently of their containing objects:

```

>>> D
{'hack': 1, 'Py': 2, 'code': 3}

>>> D['Py'] = ['app', 'dev']          # Change entry (value=list)
>>> D
{'hack': 1, 'Py': ['app', 'dev'], 'code': 3}

>>> del D['code']                     # Delete entry (also D.pop(k))
>>> D
{'hack': 1, 'Py': ['app', 'dev']}

>>> D['years'] = 32                   # Add new entry
>>> D
{'hack': 1, 'Py': ['app', 'dev'], 'years': 32}

>>> D['Py'][0] = 'program'            # Change a nested list in place
>>> D
{'hack': 1, 'Py': ['program', 'dev'], 'years': 32}

```

Like lists, assigning to an existing index in a dictionary changes its associated value. Unlike lists, however, whenever you assign a *new* dictionary key (one that isn't already present) you create a new entry in the dictionary, as was done in the previous example for the key 'years'. This doesn't work for lists because you can only assign to existing list offsets—Python considers an offset beyond the end of a list out of bounds and raises an error. As you learned earlier, to expand a list, you need to use tools such as the `append` method or slice assignment instead.

More Dictionary Methods

Dictionary methods provide a variety of type-specific tools. For instance, the `values` and `items` methods return all of the dictionary's values and (*key,value*) pair tuples, respectively; along with `keys`, these are useful in loops that need to step through dictionary entries one by one (we'll start coding such loops later in this chapter). As with `keys`, these two methods also return *iterable* objects, and wrapping them in a `list` call collects their values all at once for display:

```

>>> D = {'program': 1, 'script': 2, 'app': 3}

>>> list(D.keys())                    # All keys
['program', 'script', 'app']

>>> list(D.values())                  # All values
[1, 2, 3]

>>> list(D.items())                  # All (key, value) tuples
[('program', 1), ('script', 2), ('app', 3)]

```

In realistic programs that gather data as they run, you often won't be able to predict what will be in a dictionary before the program is launched, much less when it's coded. Fetching a nonexistent key is normally an error, but the `get` method returns a default value—`None`, or a passed-in default—if the key doesn't exist. It's an easy way to fill in a default for a key that isn't present and avoid a missing-key error when your program can't anticipate contents ahead of time:

```

>>> D.get('script')                  # A key that is present
2

>>> print(D.get('code'))              # A key that is missing
None

>>> D.get('code', 4)
4

```

The update method provides something similar to concatenation for dictionaries (though it works in the realm of keys and values, not just values). It *merges* the keys and values of one dictionary into another, both adding new entries for new keys, and blindly overwriting values of the same key if there's a clash:

```
>>> D
{'program': 1, 'script': 2, 'app': 3}
>>> D2 = {'code':4, 'hack':5, 'app': 6}    # New keys added, app:6 wins
>>> D.update(D2)
>>> D
{'program': 1, 'script': 2, 'app': 6, 'code': 4, 'hack': 5}
```

Finally, the dictionary pop method deletes a key from a dictionary and returns the value it had. It's similar to the list pop method, but it takes a key instead of an optional position:

```
>>> D.pop('app')                # Pop a dictionary key
6
>>> D.pop('hack')               # Delete key and return its value
5
>>> D
{'program': 1, 'script': 2, 'code': 4}

>>> L = ['aa', 'bb', 'cc', 'dd']    # Pop a list by position
>>> L.pop()                       # Delete and return from the end
'dd'
>>> L.pop(1)                      # Delete from a specific position
'bb'
>>> L
['aa', 'cc']
```

Dictionaries also provide a copy method that, as you might guess, makes a copy; we'll revisit this in [Chapter 9](#), as it's a way to avoid the potential side effects of shared references to the same dictionary. In fact, dictionaries come with more methods than the common ones demoed here, and over time may gain others beyond those listed in [Table 8-2](#); again, see the Python library manual, `dir` and `help`, or other reference resources for a comprehensive list.

Other Dictionary Makers

Because dictionaries are so useful, multiple ways to build them have emerged over time. The following summarizes the most common alternatives; its last two calls to the `dict` constructor (really, type name) have the same effect as the literal and key-assignment forms listed above them:

```
{'name': 'Pat', 'age': 40}        # 1) Traditional literal expression

D = {}                            # 2) Assign by keys dynamically
D['name'] = 'Pat'
D['age'] = 40

dict(name='Pat', age=40)          # 3) dict keyword-argument form

dict([('name', 'Pat'), ('age', 40)]) # 4) dict key/value tuples form
```

All four of these forms create the same two-key dictionary, but they are useful in differing circumstances:

- The first is handy if you can spell out the entire dictionary ahead of time.
- The second is of use if you need to create the dictionary one field at a time on the fly.
- The third involves less typing than the first, but it requires all keys to be strings.
- The last is useful if you need to build up keys and values as sequences at runtime.

We met *name=value* keyword arguments earlier when sorting lists; the `dict` form in this summary that uses them is newer than literals and common in Python code, because it has less syntax (and hence less room for mistakes). The last form in the summary is also commonly used in conjunction with the `zip` function, to combine separate lists of keys and values obtained dynamically at runtime (parsed out of a data file's columns, for instance):

```
dict(zip(keylist, valueslist))           # Zipped key/value tuples form (ahead)
```

There is more on zipping dictionary keys in the next section. Provided all the key's values are the same initially, you can also create a dictionary with the following special form—simply pass in a list of keys and an initial value for all of the values (the default is `None`). Notice this is run from the `dict` type name, not an actual dictionary:

```
>>> dict.fromkeys(['a', 'b'], 0)
{'a': 0, 'b': 0}
```

Although you could get by with just literals and key assignments at this point in your Python career, you'll probably find uses for all of these dictionary-creation forms as you start applying them in realistic Python programs.

Dictionary-literal unpacking

As of Python 3.5, dictionary *literals* also support a special `**` syntax that *unpacks* the contents of another dictionary in its top level, similar to the `*` in list literals we met earlier. Any number of `**` can appear in a literal to unpack any number of dictionaries, and the rightmost's value wins when keys collide:

```
>>> D = dict(a=4, c=3)
>>> {'a': 1, 'b': 2, **D}           # Dictionary-literal unpacking
{'a': 4, 'b': 2, 'c': 3}
```

```
>>> dict(a=1, b=2) | D              # Same, but with union operator
{'a': 4, 'b': 2, 'c': 3}
```

As shown, the effect of `**` is similar to both the `update` method we met earlier and the dictionary `|` union covered ahead and might avoid a follow-up `update` call in some contexts. It's also related to the extended-unpacking assignment statement you'll meet in [Chapter 11](#).

The `**` also works in `dict`, but simply because `**` unpacks keyword arguments in *any* function call (as usual in Python, this relies on larger concepts we'll reach later in this book). Unlike in `{}` literals though, `**` in `dict` fails if any key appears twice:

```
>>> dict(a=1, **{'b': 2}, **dict(c=3))   # Works in dict(), as in all calls
{'a': 1, 'b': 2, 'c': 3}                 # As long as no keys are repeated!

>>> dict(a=1, b=2, **dict(a=4, c=3))
TypeError: dict() got multiple values for keyword argument 'a'
```


The examples so far demo the many basic ways to create dictionaries, but there is yet another that's even more powerful, as the next section will explain.

Dictionary Comprehensions

Like the set and list comprehensions we met in previous coverage, dictionary comprehensions run an implied loop, collecting the key/value results of expressions on each iteration and using them to fill out a new dictionary. A loop variable allows the comprehension to use loop iteration values along the way. The net effect lets us create new dictionaries with small bits of code that are simpler than the full-blown statements we'll study later in this book.

Abstractly, dictionary comprehensions map to for loops as follows, where both *k* and *v* can use loop variable *x*:

```
{k: v for x in iterable}           # Dictionary comprehension

new = {}                           # Equivalent loop code
for x in iterable:
    new[k] = v
```

To illustrate, a standard way to initialize a dictionary dynamically is to combine its keys and values with `zip`, and pass the result to the `dict` call, per the last section. The `zip` built-in function is the hook that allows us to construct a dictionary from key and value lists this way—if you cannot predict the set of keys and values in your code, you may be able to build them up as lists and zip them together. We'll study `zip` in detail in the next part of this book; it's an iterable, so we must wrap it in a `list` call to show its results there, but its basic usage is otherwise straightforward:

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3]))    # Zip together keys and values
[('a', 1), ('b', 2), ('c', 3)]

>>> D = dict(zip(['a', 'b', 'c'], [1, 2, 3])) # Make a dict from zip result
>>> D
{'a': 1, 'b': 2, 'c': 3}
```

You can achieve the same effect, though, with a dictionary *comprehension* expression. The following uses tuple assignment to unpack items into variables (per its [Chapter 4](#) debut), as it scans the list of zipped pairs from left to right. The net effect builds a new dictionary with a key/value pair for every such pair in the `zip` result (the Python code reads almost the same as this natural-language description, but with a bit more formality):

```
>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
>>> D
{'a': 1, 'b': 2, 'c': 3}
```

Comprehensions are longer to code in this case, but they're also more general than this example implies—we can use them to map a single stream of values to dictionaries and apply them to any kind of sequence (or other iterable), and collected keys can be computed with expressions just like collected values:

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}    # Or: in range(1, 5)
>>> D
{1: 1, 2: 4, 3: 9, 4: 16}

>>> D = {c: c * 4 for c in 'HACK'}          # Loop over any iterable
>>> D
{'H': 'HHHH', 'A': 'AAAA', 'C': 'CCCC', 'K': 'KKKK'}
```

```
>>> D = {c.lower(): (c + '!') for c in ['HACK', 'PY', 'CODE']}    # Expr: expr
>>> D
{'hack': 'HACK!', 'py': 'PY!', 'code': 'CODE!'}
```

Dictionary comprehensions are also useful for initializing dictionaries from keys lists, in much the same way as the `fromkeys` method we met at the end of the preceding section:

```
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)    # Initialize dict from keys
>>> D
{'a': 0, 'b': 0, 'c': 0}

>>> D = {k: 0 for k in ['a', 'b', 'c']}    # Same, but with a comprehension
>>> D
{'a': 0, 'b': 0, 'c': 0}

>>> D = dict.fromkeys('code')    # Other iterables, default value
>>> D
{'c': None, 'o': None, 'd': None, 'e': None}

>>> D = {k: None for k in 'code'}
>>> D
{'c': None, 'o': None, 'd': None, 'e': None}
```

Like its list and set relatives, dictionary comprehensions support additional syntax not shown here, including `if` clauses to filter values out of results, and nested `for` loops. Unfortunately, to truly understand dictionary comprehensions, you need to also know more about iteration statements and concepts in Python, and this book hasn't yet disclosed enough information to tell that story well. You'll learn much more about all flavors of comprehensions—list, set, dictionary, and generator—in [Chapters 14](#) and [20](#), so we'll defer further details until later.

Key Insertion Ordering

By now, you've probably noticed that dictionaries remember their keys' order—no matter what sort of syntax we use to make them. As noted both earlier here and in [Chapter 4](#), as of Python 3.7 (and CPython 3.6), dictionaries have shed their former random key order and adopted *insertion order* instead: keys are now ordered left to right from oldest to newest additions. This order is maintained for literals, `dict` calls, comprehensions, and new keys added on the fly.

Among other things, insertion order makes dictionary outputs arguably more coherent and readable and may avoid key sorts in some programs. This isn't the same as a list (e.g., there's no way to add a key in the “middle”) and doesn't make dictionaries sequences. The effect, however, is close. Here's a brief illustration:

```
>>> D = dict(a=1, b=2)    # Literal keys stored left-to-right
>>> D['c'] = 3    # New keys always added at the end
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> D.pop('b')    # Method pop removes any key by name
2
>>> D    # Other keys still in insertion order
{'a': 1, 'c': 3}
>>> D['b'] = 2    # Earlier key goes at the end too
>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> D['c'] = 0    # Changing values doesn't impact order
>>> D['d'] = 1    # And new arrivals always go at the end
>>> D
{'a': 1, 'c': 0, 'b': 2, 'd': 1}
```

If you stare at this long enough, you'll probably notice that the insertion order of dictionaries is much like the LIFO *stack* ordering we coded with list `append` and `pop` methods earlier. In fact, the dictionary `popitem` is defined to return the key/value pair (really, tuple) for the key added most recently (i.e., at the end):

```
>>> D.popitem()
('d', 1)
>>> D.popitem()
('b', 2)
>>> D
{'a': 1, 'c': 0}
```

While useful in some contexts, keep in mind that you may still need to *sort* dictionary keys manually, using techniques we'll study ahead. Ordering filenames for display, for example, will often warrant string-value sorts instead of a program's internal insertion order.

Dictionary “Union” Operator

Last up in the dictionary toolbox is a bit of an oddball: as of Python 3.9, dictionaries have sprouted a “union” operation that's kicked off with the `|` operator—the same syntax used for union in the set objects you met in [Chapter 5](#).

Per the quotes, though, this operation isn't really mathematical set union at all. It's a positionally sensitive, key-based *merge* of dictionaries, that works the same as the dictionary `update` method but *returns* its result as a new dictionary instead of updating a subject in place. Hence, dictionary `|` is like the combination of running the dictionary's `copy` and `update` methods in turn, and really just a minor convenience for narrow roles. Here it is at work:

```
>>> D = dict(a=1, b=2)
>>> D
{'a': 1, 'b': 2}

>>> D | {'b': 3, 'c': 4}           # Update, return, rightmost wins
{'a': 1, 'b': 3, 'c': 4}

>>> D | {'b': 3, 'c': 4} | dict(a=5, d=6)
{'a': 5, 'b': 3, 'c': 4, 'd': 6}
```

You get roughly the same mileage with the dictionary's longstanding `update` method, though it requires a manual copy to avoid in-place changes in programs that don't want them (as a spoiler, the `|=` in-place assignment form of dictionary union that you'll meet later in this book is *identical* to calling `update` with a single argument sans `copy`):

```
>>> D
{'a': 1, 'b': 2}
>>> C = D.copy()
>>> C.update({'b': 3, 'c': 4})      # "Union" sans the | expression
>>> C.update(dict(a=5, d=6))
>>> C
{'a': 5, 'b': 3, 'c': 4, 'd': 6}
```

The Python 3.5 dictionary-literal *unpacking* we explored earlier can have the same effect too, though, at least when used with gnarly embedded literals as in the following, perhaps less readably:

```
>>> D
{'a': 1, 'b': 2}
>>> {**D, **{'b': 3, 'c': 4}, **dict(a=5, d=6)}
{'a': 5, 'b': 3, 'c': 4, 'd': 6}
```

Which—true sports fans will note—raises the number of dedicated dictionary merge operations from one to *three* since this book’s prior edition. This bloat is especially grievous, given that it takes *just one line of simple code* to accomplish what the `update` method, the newer `**` unpacking, and the newest `|` union operator all redundantly do in their main use cases:

```
>>> D1 = dict(a=1, b=1)
>>> D2 = dict(a=2, c=2)

>>> for k in D2: D1[k] = D2[k]           # What update(), **, and | redundantly do

>>> D1                                   # Did this really justify 3 alternatives?
{'a': 2, 'b': 1, 'c': 2}
```

Dictionary `|` “union” may be handy in limited contexts. But whether this justifies convoluting dictionaries with just one binary operator, just one of many set operations, and an operation that’s almost entirely redundant with two earlier tools that were already almost entirely redundant with very simple code, is a riddle left to the reader to solve.

Intermission: Books Database

Let’s take a break from the fine points and code a more concrete example of dictionaries at work. In honor of this book’s quarter-century milestone, the following builds a simple in-memory editions database that maps edition *years* (the keys) to *titles* (the values). As coded, you fetch edition names by indexing on year strings:

```
>>> table = {'2024': 'Learning Python, 6th Edition',      # Year => title
             '2013': 'Learning Python, 5th Edition',
             '1999': 'Learning Python'}

>>> table['2024']                                         # Key => Value
'Learning Python, 6th Edition'

>>> for year in table:                                   # Keys iteration
    print(year + '\t' + table[year])

2024      Learning Python, 6th Edition
2013      Learning Python, 5th Edition
1999      Learning Python
```

The last command uses a `for` loop, which we’ve used several times since its [Chapter 4](#) preview. The full tale of the `for` isn’t told until [Chapter 13](#), but this particular loop simply iterates through each key in the table to print a tab-separated list of keys and their values (recall from [Chapter 7](#) that `\t` in a Python string means vertical tab).

Dictionaries aren’t sequences like lists and strings, but if you need to step through the items in a dictionary, it’s easy—either call the dictionary `keys` method or use the dictionary itself. Given a dictionary `D`, saying `for key in D` works the same as saying `for key in D.keys()`, and both use the *iteration protocol* that we’ll expand on later in this book. Either way, you can index from *key* to *value* inside the `for` loop as you go, as this code does.

Mapping values to keys

Notice how the prior table maps years to titles, but not vice versa. If you want to map the other way—titles to years—you can either code the dictionary differently:

```
>>> table = {'Learning Python, 6th Edition': 2024,          # Title => year
             'Learning Python, 5th Edition': 2013,
             'Learning Python': 1999}

>>> table['Learning Python']                                # Key => value
1999
```

Or use other dictionary methods like `items` that give searchable sequences (the `list` call is required in the following because `items` returns an iterable *view*, a topic coming up shortly):

```
>>> list(table.items())[:2]
[('Learning Python, 6th Edition', 2024), ('Learning Python, 5th Edition', 2013)]

>>> [title for (title, year) in table.items() if year == 1999]    # Value => key
['Learning Python']
```

The last command here uses the list *comprehension* we explored earlier, as well as *tuple assignment* mentioned earlier and covered in this book's next part (synopsis: it unpacks items into variables). The combo scans the (*key*, *value*) pairs returned by the dictionary's `items` method, selecting keys for values matching a search target (1999).

The net effect of all this is to index *backward*—from value to key, instead of key to value. This is useful if you want to store data just once and map backward from values only rarely (searching through sequences like this is generally much slower than a direct key-to-value index, though not all programs need to care).

In fact, although dictionaries by nature map keys to values unidirectionally, there are multiple ways to map values back to keys with a bit of extra generalizable code:

```
>>> K = 'Learning Python'
>>> V = 1999
>>> table[K]          # Key => Value (normal usage)
1999

>>> [key for (key, value) in table.items() if value == V]        # Value => Key
['Learning Python']

>>> [key for key in table.keys() if table[key] == V]             # Same: keys()
['Learning Python']
```

Note that both of the last two commands return a *list* of titles: in dictionaries, there's just *one* value per key, but there may be *many* keys per value if a given value is be stored under multiple keys, and a value might be a collection itself to represent many values per key. It's also possible to *invert* a dictionary for indexing values by zipping its values and keys—but *only* if its values are all immutable and don't appear in multiple keys:

```
>>> dict(zip(table.values(), table.keys()))[1999]                # Key/value inversion?
'Learning Python'
```

Sharp-eyed readers may notice that this yields a dictionary with *integer* keys, which works naturally per a tip in the next section. For more on this front, watch for a less lossy dictionary inversion function in the *mapattrs.py* example from “[Example: Mapping Attributes to Inheritance Sources](#)” on page 811—code that would surely stretch this preview past its breaking point if included here. For this chapter's purposes, let's wrap up by adding in a few pragmatic pieces to the dictionary puzzle.

Dictionary Usage Tips

Dictionaries are fairly straightforward tools once you get the hang of them, but here are a few additional pointers and reminders you should be aware of when using them:

Sequence operations don't work

Dictionaries are mappings, not sequences. Because they deal with keys and values and are ordered by insertion-time only, things like concatenation (an ordered joining of values) and slicing (extracting a contiguous section of values) simply don't apply—and Python raises an error if your code tries to use them on dictionaries.

Assigning to new indexes adds entries

Keys can be created when you write a dictionary literal (embedded in the code of the literal itself), or when you assign values to new keys of an existing dictionary object individually. The end result is the same.

Keys need not always be strings

Our examples so far have used strings as keys, but any other *immutable* objects work just as well. For instance, you can use integers as keys, which makes the dictionary look much like a list (when indexing, at least). Tuples may be used as dictionary keys too, allowing compound key values—such as dates and IP addresses—to have associated values. User-defined class instance objects (discussed in [Part VI](#)) can also be used as keys, as long as they define the proper methods; roughly, they need to tell Python that their values are “hashable” and thus won't change, as otherwise they would be useless as fixed keys. Mutable objects such as lists, sets, and other dictionaries don't work as keys because they may change, but are allowed as values.

The following sections delve deeper into these and other mysteries of dictionary-processing code.

Using dictionaries to simulate flexible lists: Integer keys

The last point in the preceding list is important enough to demonstrate with a few examples. When you use lists, it is illegal to assign to an offset that is off the end of the list:

```
>>> L = []
>>> L[99] = 'hack'
IndexError: list assignment index out of range
```

Although you can use repetition to preallocate as big a list as you'll need (e.g., `[0]*100`), you can also do something that looks similar with dictionaries that does not require such space allocations—and potential waste. By using integer keys, dictionaries can emulate lists that seem to grow on offset assignment:

```
>>> D = {}
>>> D[99] = 'hack'
>>> D[99]
'hack'
>>> D
{99: 'hack'}
```

Here, it looks as if `D` is a 100-item list, but it's really a dictionary with a single entry; the value of the key 99 is the string `'hack'`. You can access this structure with offsets much like a list, catching nonexistent keys with `get` or `in` tests if required, but you don't have to allocate space for all the positions to which you might need to assign values in the future. When used like this, dictionaries are like more flexible equivalents of lists:

```
>>> D[62] = 'code'
>>> D[30] = 'write'
>>> D[62]
'code'
>>> D
{99: 'hack', 62: 'code', 30: 'write'}
```

As another example, we might also employ integer keys in the first book *book-database* code we wrote earlier to avoid quoting the year, albeit at the expense of expressiveness (integer keys cannot contain nondigit characters):

```
>>> table = {2024: 'Learning Python, 6th Edition',      # Integers work as keys
...etc...
>>> table[2024]
'Learning Python, 6th Edition'
```

Using dictionaries for sparse data structures: Tuple keys

In a similar way, dictionary keys are also commonly leveraged to implement *sparse*—mostly empty—data structures, such as multidimensional arrays where only a few positions have values stored in them:

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4                # A ; separates statements: see Chapter 10
>>> Matrix[(X, Y, Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}
```

Here, we’ve used a dictionary to represent a three-dimensional array that is empty except for the two positions (2,3,4) and (7,8,9). The keys are *tuples* that record the coordinates of nonempty slots. Rather than allocating a large and mostly empty three-dimensional matrix to hold these values, we can use a simple two-item dictionary. In this scheme, accessing an empty slot triggers a nonexistent key exception, as these slots are not physically stored:

```
>>> Matrix[(2,3,6)]
KeyError: (2, 3, 6)
```

Avoiding missing-key errors

Errors for nonexistent key fetches like the foregoing are common in sparse matrixes, but you probably won’t want them to shut down your program. There are at least three ways to fill in a default value instead of getting such an error message—you can use the dictionary `get` method shown earlier to provide a default for keys that do not exist, test for keys ahead of time in `if` statements, or use a `try` statement to catch and recover from the error. Though straightforward, the last of these are previews of statement syntax we’ll begin studying in [Chapter 10](#):

```
>>> Matrix.get((2, 3, 4), 0)          # Exists: fetch and return
88
>>> Matrix.get((2, 3, 6), 0)          # Doesn't exist: use passed default
0

>>> if (2, 3, 6) in Matrix:           # Check for key before fetch
...     print(Matrix[(2, 3, 6)])      # See Chapters 10 and 12 for if/else
... else:
...     print(0)
...
```

0

```
>>> try:
...     print(Matrix[(2, 3, 6)])          # Try to index
... except KeyError:                     # Catch and recover
...     print(0)                         # See Chapters 10 and 34 for try/except
...
0
```

Of these, the `get` method is the most concise in terms of coding requirements, but the `if` and `try` statements are much more general in scope—as you’ll start seeing for yourself soon in [Chapter 10](#).

Nesting in dictionaries

As you can tell, dictionaries can play many roles in Python. In general, they can replace search data structures (because indexing by key is a search operation) and can represent many types of structured information. For example, dictionaries are one of many ways to describe the properties of an item in your program’s domain; that is, they can serve the same role as “records” or “structs” in other language, and *JSON* content in language-neutral roles. The following, for example, fills out a dictionary describing a book, by assigning to new keys:

```
>>> rec = {}
>>> rec['title'] = 'Learning Python, 5th Edition'
>>> rec['year'] = 2013
>>> rec['isbn'] = '9781449355739'
>>>
>>> rec['year'], rec['isbn']
(2013, '9781449355739')
```

Especially when nested, though, Python’s built-in data types allow us to easily represent *structured* information. The following again uses a dictionary to capture object properties, but it codes it all at once rather than assigning to each key separately, and nests a dictionary and list to represent structured property values:

```
>>> rec = {'title': 'Learning Python, 5th Edition',
          'date': {'year': 2013, 'month': 'July'},
          'isbns': ['1449355730', '9781449355739']}
```

To fetch components of nested objects, simply string together indexing operations:

```
>>> rec['title']
'Learning Python, 5th Edition'
>>> rec['isbns']
['1449355730', '9781449355739']
>>> rec['isbns'][1]
'9781449355739'
>>> rec['date']['year']
2013
```

Although you’ll learn in [Part VI](#) that *classes* (which group both data and logic) can sometimes be better in this record role, dictionaries are an easy-to-use tool for simpler requirements. For more on record representation choices, see also the upcoming sidebar “[Why You Will Care: List Versus Dictionary Versus Set](#)” on page 204.

Also notice that while we’ve focused on a single “record” with nested data here, there’s no reason we couldn’t nest the record itself in a larger, enclosing *database* collection coded as a list or dictionary, though an external file or formal database interface often plays the role of top-level container in realistic

programs. The following abstract snippets would both print a record's two-item `isbn` list if run live and provided with an other record omitted here:

```
db = []
db.append(rec)           # A list "database"
db.append(other)
db[0]['isbn']

db = {}
db['lp5e'] = rec         # A dictionary "database"
db['lp6e'] = other
db['lp5e']['isbn']
```

Later in the book you'll meet tools such as Python's `shelve`, which works much the same way, but automatically maps objects to and from files to make them permanent. Python objects really *can* be database records.

Dictionary key/value/item view objects

When we explored the dictionary `keys`, `values`, and `items` methods earlier, we wrapped their results in list calls for display at the REPL prompt. Technically, this is because these methods return *view objects* that produce results on demand, instead of physical lists. Displaying their raw values suggests as much, but collects their values anyhow:

```
>>> D = dict(program=1, script=2, app=3)
>>> D.keys()
dict_keys(['program', 'script', 'app'])
```

View objects are *iterables*, which we've seen simply means objects that generate result items one at a time, instead of producing the result list all at once in memory. Besides being iterable, dictionary views are *insertion* ordered, reflect future *changes* to the dictionary, and support *set* operations. On the other hand, because they are not lists, they do not directly support operations like indexing or the list `sort` method, and do not display as a normal list when printed.

We'll discuss the notion of iterables more formally in [Chapter 14](#), but for our purposes here it's enough to know that we have to run the results of these three methods through the `list` built-in if we want to apply list operations or display their values as lists. For example:

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'b': 2, 'c': 3}

>>> K = D.keys()           # Makes a view object, not a list
>>> K
dict_keys(['a', 'b', 'c'])
>>> list(K)                # Force a real list when needed
['a', 'b', 'c']

>>> V = D.values()         # Ditto for values and items views
>>> V
dict_values([1, 2, 3])
>>> list(V)
[1, 2, 3]

>>> D.items()
dict_items([('a', 1), ('b', 2), ('c', 3)])
>>> list(D.items())
[('a', 1), ('b', 2), ('c', 3)]
```

```
>>> K[0]                                # List operations fail unless converted
TypeError: 'dict_keys' object is not subscriptable
>>> K.sort()
AttributeError: 'dict_keys' object has no attribute 'sort'
>>> list(K)[0], list(K).sort()
('a', None)
```

Unlike lists, though, dictionary views don't take up *space* for their full results, and are not carved in stone when created—they *dynamically reflect future changes* made to the dictionary after the view object has been created:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'b': 2, 'c': 3}

>>> K = D.keys()
>>> V = D.values()
>>> list(K), list(V)                    # Views maintain same order as dictionary
(['a', 'b', 'c'], [1, 2, 3])

>>> del D['b']                          # Change the dictionary in place
>>> D
{'a': 1, 'c': 3}

>>> list(K), list(V)                    # Reflected in any current view objects
(['a', 'c'], [1, 3])
```

That said, use cases for dynamic view morph are likely rare. In fact, apart from result displays at the interactive prompt, you probably won't even notice views very often in practice, because looping constructs in Python automatically force them to produce one result on each:

```
>>> for k in D.keys(): print(k)         # View iterators used automatically in loops
...                                     # But no need to call keys() to iterate on D
a
b
c
```

As we've seen, it's also often unnecessary to call keys directly in such cases because dictionaries themselves provide *implicit* key iterators; for better or worse, `for k in D` is usually as much code as you need to type.

Dictionary views and sets

Though perhaps even more obscure than view objects in general, some dictionary views are also *set-like* and support set operations such as union and intersection that we used on true sets in [Chapter 5](#). Specifically, views returned by the keys method are set-like, values views are not, and items views are if their (*key*, *value*) pairs are unique and hashable (immutable). This reflects logical symmetry: set items are unique and immutable just like dictionary keys, and sets themselves behave like unordered and valueless dictionaries (and are even coded in curly braces).

Here is what keys views look like when used in set operations (continuing the prior section's session); as also shown, dictionary value views are never set-like, because their items are not necessarily unique or immutable:

```
>>> K, V
(dict_keys(['a', 'c']), dict_values([1, 3]))

>>> K | {'x': 4}                        # Keys (and some items) views are set-like
{'x', 'a', 'c'}                        # Results are unordered sets, not views
```

```
>>> V | {'x': 4}
TypeError: unsupported operand type(s) for |: 'dict_values' and 'dict'
>>> V | {'x': 4}.values()
TypeError: unsupported operand type(s) for |: 'dict_values' and 'dict_values'
```

Be careful not to confuse the `|` here with the *dictionary* union operation we met earlier in “[Dictionary “Union” Operator](#)” on page 193; when run on a view, `|` is not a full key-based dictionary merge. In set operations, views may be mixed with other views, sets, and dictionaries, and dictionaries are treated the same as their keys views:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D.keys() & D.keys()           # Intersect views
{'b', 'a', 'c'}
>>> D.keys() & {'b'}             # Intersect view and set
{'b'}
>>> D.keys() & {'b': 1}          # Intersect view and dict
{'b'}
>>> D.keys() | {'b', 'c', 'd'}   # Union view and set
{'b', 'a', 'd', 'c'}
```

Items views are set-like too, but only if they are *hashable*—that is, if they contain only immutable objects:

```
>>> {'a': [1, 2]}.items() | {0, 1}   # Immutable value: no set ops
TypeError: unhashable type: 'list'

>>> D = {'a': 1}
>>> D.items()                       # Items set-like if and only if hashable
dict_items([('a', 1)])
>>> D.items() | D.keys()             # Union view and view
{('a', 1), 'a'}
>>> D.items() | D                  # Dictionary treated same as its keys
{('a', 1), 'a'}

>>> D.items() | {('c', 3), ('d', 4)} # Set of key/value pairs
{('a', 1), ('c', 3), ('d', 4)}

>>> dict(D.items() | {('c', 3), ('d', 4)}) # dict accepts iterable sets too
{'a': 1, 'c': 3, 'd': 4}
```

See [Chapter 5](#)’s coverage of sets if you need a refresher on these operations. Their role in dictionary views is probably uncommon and may even be academic; but they work when helpful (and at least you now have a fighting chance when they crop up in sadistic final exams!). Here, let’s wrap up with two more quick coding notes for dictionaries.

Sorting dictionary keys

First of all, keys lists must be sorted when the inherent insertion order doesn’t meet your goals. Beware, though, that a common coding pattern for scanning a collection in sorted order won’t work, because `keys` does not return a list:

```
>>> D = {'c': 3, 'b': 2, 'a': 1}
>>> D
{'c': 3, 'b': 2, 'a': 1}

>>> Ks = D.keys()                  # Sorting a view object doesn't work!
>>> Ks.sort()
AttributeError: 'dict_keys' object has no attribute 'sort'
```

To work around this, you can either convert keys to a list manually, or use the `sorted` call, applied to lists earlier in this chapter, on either a keys view or the dictionary itself:

```
>>> Ks = list(D.keys())           # Convert keys to a list
>>> Ks.sort()                     # And then sort with list.sort()
>>> for k in Ks: print(k, D[k])
...
a 1
b 2
c 3

>>> Ks = D.keys()                 # Or use sorted() on the keys view
>>> for k in sorted(Ks): print(k, D[k]) # sorted() accepts any iterable
...                               # sorted() returns its result
a 1
b 2
c 3
```

Of these, using the dictionary's keys iterator is simplest and common, though also arguably implicit:

```
>>> for k in sorted(D): print(k, D[k]) # Or use sorted() on the dict
...                                   # dict iterators produce keys
a 1
b 2
c 3
```

Dictionary magnitude comparisons

Finally, dictionaries can be compared for equality directly with `==`, which automatically compares each key/value pair and ignores insertion order:

```
>>> D1 = dict(a=1, b=2, c=3)
>>> D2 = dict(c=3, b=2, a=1)
>>> D1, D2
({'a': 1, 'b': 2, 'c': 3}, {'c': 3, 'b': 2, 'a': 1})
>>> D1 == D2, D1 != D2
(True, False)
```

Magnitude comparisons like `<` and `>` do not work on dictionaries themselves, though you can implement them by comparing key/value items views manually, if you also *sort* them to discount any insert-order difference (subtlety: a `>` on the items results directly runs the set *superset* test because the views are set-like—not greater-than!):

```
>>> D1 = dict(a=1, b=2, c=4)
>>> D2 = dict(c=3, b=2, a=1)           # D1 > D2: D1['c'] > D2['c']
>>> D1 > D2
TypeError: '>' not supported between instances of 'dict' and 'dict'

>>> list(D1.items()), list(D2.items())
[('a', 1), ('b', 2), ('c', 4)], [('c', 3), ('b', 2), ('a', 1)]

>>> list(D1.items()) > list(D2.items()) # Fails: insertion order
False
>>> D1.items() > D2.items()           # Fails: superset, not magnitude
False
>>> sorted(D1.items()) > sorted(D2.items()) # OK: sorted to neutralize order
True
```

Because we'll revisit this near the end of the next chapter to reinforce them in the context of comparisons at large, we'll postpone further coverage here.

Chapter Summary

In this chapter, we explored the list and dictionary types—probably the two most common, flexible, and powerful collection types you will see and use in Python code. We learned that the list type supports positionally ordered collections of arbitrary objects, and that it may be freely nested and grown and shrunk on demand. The dictionary type is similar, but it stores items by key instead of by position and orders keys by insertion time only. Both lists and dictionaries are mutable, and so support a variety of in-place change operations not available for strings: for example, lists can be grown by slice assignment and append calls, and dictionaries by key assignment and update.

In the next chapter, we will wrap up our in-depth core object-type tour by looking at tuples and files. After that, we'll move on to statements that code the logic that processes our objects, taking us another step toward writing complete programs. Before we tackle those topics, though, here are some chapter quiz questions to review what you've learned.

Test Your Knowledge: Quiz

1. Name two ways to build a list containing five integer zeros.
2. Name two ways to build a dictionary with two keys, 'a' and 'b', each having an associated value of 0.
3. Name four operations that change a list object in place.
4. Name four operations that change a dictionary object in place.
5. Why might you use a dictionary instead of a list?

Test Your Knowledge: Answers

1. A literal expression like `[0, 0, 0, 0, 0]` and a repetition expression like `[0] * 5` will each create a list of five zeros. In practice, you might also build one up with a loop that starts with an empty list and appends 0 to it in each iteration, with `L.append(0)`. A list comprehension like `[0 for i in range(5)]` could work here, too, but this is more work than you need to do for this answer.
2. A literal expression such as `{'a': 0, 'b': 0}` or a series of assignments like `D = {}`, `D['a'] = 0`, and `D['b'] = 0` would create the desired dictionary. You can also use the newer and simpler-to-code `dict(a=0, b=0)` keyword form, or the more flexible `dict([('a', 0), ('b', 0)])` key/value sequences form. Because all the values are the same, you can also use the special form `dict.fromkeys('ab', 0)`, and dictionary comprehension like `{k: 0 for k in 'ab'}` suffices too, though again, this may be overkill here.
3. The `append` and `extend` methods grow a list in place, the `sort` and `reverse` methods order and reverse lists, the `insert` method inserts an item at an offset, the `remove` and `pop` methods delete from a list by value and by position, the `del` statement deletes an item or slice, and index and slice assignment statements replace an item or entire section. Pick any four of these for the quiz.
4. Dictionaries are primarily changed by *assignment* to a new or existing key, which creates or changes the key's associated value in the table. Also, the `del` statement deletes a key's entry, the dictionary update method merges one dictionary into another in place, and `D.pop(key)` removes a key and returns the value it had. Dictionaries also have other, more exotic in-place change

methods presented tersely or not at all in this chapter, such as `popitem` and `setdefault`; see reference resources for more details.

5. This question is a bit unfair, given that the following sidebar gives the answer, but you may have already figured this out on your own. Dictionaries are generally better when the data is labeled (a record with field names, for example); lists are best suited to collections of unlabeled items (such as all the files in a directory). Dictionary lookup is also usually quicker than searching a list, though this might vary per program and Python.

Why You Will Care: List Versus Dictionary Versus Set

With all the objects in Python's core types arsenal, some readers may be puzzled over the choice between lists and dictionaries. In short, although both are flexible collections of other objects, lists assign items to *positions*, and dictionaries assign them to more mnemonic *keys*. Because of this, dictionary data often carries more meaning to human readers. For example, a nested list structure can always be used to record info:

```
>>> L = ['Pat', 40.5, ['dev', 'mgr']] # List-based "record"
>>> L[0]
'Pat'
>>> L[1]                                # Positions/numbers for fields
40.5
>>> L[2][1]
'mgr'
```

For some types of data, the list's access-by-position makes sense—a list of employees in a company, the files in a directory, or numeric matrixes, for example. But a more symbolic record like this may be more meaningfully coded as a dictionary, with labeled fields replacing field positions:

```
>>> D = {'name': 'Pat', 'age': 40.5, 'jobs': ['dev', 'mgr']}
>>> D['name']
'Pat'
>>> D['age']                                # Dictionary-based "record"
40.5
>>> D['jobs'][1]                            # Names mean more than numbers
'mgr'
```

For variety, here is the same record recoded with `dict` and keywords, which may seem even more readable to some human readers:

```
>>> D = dict(name='Pat', age=40.5, jobs=['dev', 'mgr'])
>>> D['name']
'Pat'
>>> D['jobs'].remove('mgr')
>>> D
{'name': 'Pat', 'age': 40.5, 'jobs': ['dev']}
```

In practice, dictionaries tend to be best for data with labeled components, as well as structures that can benefit from quick, direct lookups by name, instead of slower linear (left-to-right) searches. As we've seen, they also may be better for sparse collections and collections that grow flexibly.

Python programmers also have access to the *sets* we studied in [Chapter 5](#), which are much like the keys of a valueless dictionary; they don't map keys to values, but can often be used like dictionaries for fast lookups when there is no associated value, especially in search routines:

```
>>> D = {}
>>> D['state1'] = True           # A visited-state dictionary
>>> 'state1' in D
True
>>> S = set()
>>> S.add('state1')             # Same, but with sets
>>> 'state1' in S
True
```

Watch for a rehash of this record representation thread in the next chapter, where you'll see how *tuples* and *named tuples* compare to dictionaries in this role, as well as in [Chapter 27](#), where you'll learn how user-defined *classes* factor into this picture, combining both data and logic to process it.

Tuples, Files, and Everything Else

This chapter rounds out our in-depth tour of the core object types in Python by exploring the *tuple*, a collection of other objects that cannot be changed, and the *file*, an interface to external files on your computer. As you'll see, the tuple is a relatively simple object that largely performs operations you've already learned about for strings and lists. The file object is a commonly used and full-featured tool for processing files on a host of devices. Because files are so pervasive in programming, the basic overview of files here is supplemented by larger examples in later chapters.

This chapter also concludes this part of the book by summarizing properties common to all the core object types we've met—the notions of equality, comparisons, object copies, and so on. We'll also briefly explore other object types in Python's toolbox, including the `None` placeholder and the `namedtuple` hybrid; as you'll see, although we've covered all the primary built-in types, the object story in Python is broader than implied thus far. Finally, we'll close this part of the book by taking a look at a set of common object type pitfalls and exploring some exercises that will allow you to experiment with and cement the ideas you've learned.

One logistics note up front: as for strings in [Chapter 7](#), our exploration of files here will be limited to fundamentals that most Python programmers—and especially Python newcomers—need to know. In particular, *Unicode* text files were previewed in [Chapter 4](#), but we're going to postpone full coverage of them until [Chapter 37](#), as optional or deferred reading. For this chapter's purpose, we'll assume that the contents of any text files will be encoded and decoded per your platform's default Unicode encoding (and you won't yet need to know what that means). The basics you'll learn here, though, will apply both to the simpler files in this chapter as well as their extensions in [Chapter 37](#).

Tuples

The last collection type in our survey is the Python *tuple*. Tuples construct simple groups of objects. They work much like lists, except that tuples can't be changed in place (they're immutable) and are usually written as a series of items in parentheses, not square brackets. Although they don't support as many methods, tuples share most of their properties with lists. Here's a quick look at the basics. Tuples are:

Ordered collections of arbitrary objects

Like strings and lists, tuples are positionally ordered collections of objects (i.e., they maintain a left-to-right order among their contents). Like lists and dictionaries, they can embed any kind of object.

Accessed by offset

Like strings and lists, items in a tuple are accessed by offset (not by key); they support all the offset-based access operations, such as indexing and slicing.

Of the category “immutable sequence”

Like strings and lists, tuples are sequences; they support many of the same operations. However, like strings, tuples are immutable; they don’t support any of the in-place change operations applied to lists.

Fixed-length, heterogeneous, and arbitrarily nestable

Because tuples are immutable, you cannot change the size of a tuple without making a copy. On the other hand, tuples may contain any type of object, including other collection objects (e.g., lists, dictionaries, and other tuples), and so support arbitrary nesting.

Arrays of object references

Like lists, tuples are best thought of as object reference arrays: tuples store access points to other objects (references), and indexing a tuple is relatively quick.

Table 9-1 highlights common tuple operations. In it, T means a tuple. As shown, a tuple is written as a series of objects (technically, expressions that generate objects), separated by commas and normally enclosed in parentheses. An empty tuple is just a parentheses pair with nothing inside.

Table 9-1. Common tuple literals and operations

Operation	Interpretation
()	An empty tuple
T = (0,)	A one-item tuple (not an expression)
T = (0, 'Py', 1.2, 3)	A four-item tuple
T = 0, 'Py', 1.2, 3	Another four-item tuple (same as prior line)
T = ('Pat', ('dev', 'mgr'))	Nested tuples
T = tuple('hack')	Tuple of items in an iterable
T[i]	Index, index of index, slice, length
T[i][j]	
T[i:j]	
len(T)	
T1 + T2	Concatenate, repeat
T * 3	
T1 > T2, T1 == T2	Comparisons: magnitude, equality
'code' in T	Membership, iteration
for x in T: print(x)	
[x ** 2 for x in T]	
T = (*x, 0, *y, *x)	Iterable unpacking
T.index('Py')	Methods: search, count
T.count('Py')	
namedtuple('Emp', ['name', 'jobs'])	Named-tuple extension type

Tuples in Action

As usual, let's start an interactive session to explore tuples at work. Notice in [Table 9-1](#) that tuples do not have most of the methods that lists have (e.g., an `append` call won't work here). They do, however, support the usual sequence operations that we explored for both strings and lists, compare recursively as usual, and support the same `*` iterable-unpacking syntax in their literals that we used for lists in the preceding chapter:

```
$ python3                                # Start your REPL
>>> (1, 2) + (3, 4)                      # Concatenation
(1, 2, 3, 4)

>>> (1, 2) * 4                            # Repetition
(1, 2, 1, 2, 1, 2, 1, 2)

>>> T = (1, 2, 3, 4)
>>> T[0], T[1:3]                          # Indexing, slicing
(1, (2, 3))

>>> T == (1, 2, 3, 4), T > (1, 2, 3, 3), T > (1, 2, 3)
(True, True, True)

>>> L = ['code', 'hack']
>>> (*L, 1, 2, *(3, 4))                  # Iterable unpacking
('code', 'hack', 1, 2, 3, 4)
```

Tuple syntax peculiarities: Commas and parentheses

The second and fourth entries in [Table 9-1](#) merit a bit more explanation. Because parentheses can also enclose expressions (see [Chapter 5](#)), you need to do something special to tell Python when a single object in parentheses is a tuple object and not a simple expression. If you really want a single-item tuple, simply add a trailing comma after the single item, before the closing parenthesis:

```
>>> x = (40)                             # An integer!
>>> x
40
>>> y = (40,)                            # A tuple containing an integer
>>> y
(40,)
```

As a special case, Python also allows you to *omit* the opening and closing parentheses for a tuple in contexts where it isn't syntactically ambiguous to do so. For instance, the fourth line of [Table 9-1](#) simply lists four items separated by commas. In the context of an assignment statement, Python recognizes this as a tuple, even though it doesn't have parentheses. That's why all the comma-separated items we've typed at the REPL print with parentheses—it's a tuple:

```
>>> 1, 2, 3, 4                           # Tuple sans parentheses, in REPL and elsewhere
(1, 2, 3, 4)
```

This syntactic trick is also commonly leveraged by the *sequence assignment* shorthand we used briefly in [Chapter 7](#) and will study in earnest in [Chapter 11](#)—names on the left are paired with values on the right and assigned by position, but both sides are really tuples without parentheses:

```
>>> a, b, c = 1, 2, 3                    # Sequence assignment: tuples on both sides
>>> a, b, c
(1, 2, 3)
```

Now, some people will tell you to always use parentheses in your tuples, and some will tell you to never use parentheses in tuples (and still others have lives and won't tell you what to do with your tuples!). The most common places where the parentheses are *required* for tuple literals are those where:

- *Parentheses* matter—within a function call, or nested in a larger expression
- *Commas* matter—within a function call, or embedded in the literal of a larger object like a list or dictionary

In most other contexts, the enclosing parentheses are optional. For beginners, the best advice is that it's probably easier to use the parentheses than it is to remember when they are optional or required. Many programmers also find that parentheses tend to aid script readability by making the tuples more explicit and obvious.

And for language lawyers in the audience, bear in mind that the comma is really a sort of *lowest precedence operator*, though only in contexts where it's not otherwise significant. In such contexts, it's the *comma* that builds tuples, not the parentheses. This makes the latter optional, but can also lead to odd, unexpected syntax errors if parentheses are omitted (e.g., in `lambda` covered in [Part IV](#)). Adding parentheses to your tuples as a habit avoids the oddities.

Conversions, methods, and immutability

Apart from literal-syntax differences, tuple operations (the middle rows in [Table 9-1](#)) are identical to string and list operations. The only differences worth noting are that the `+`, `*`, and slicing operations return new *tuples* when applied to tuples, and that tuples don't provide the same methods you saw for strings, lists, and dictionaries. If you want to *sort* a tuple, for example, you'll usually have to either first convert it to a list to gain access to a sorting method call and make it a mutable object, or use the newer sorted built-in that accepts any sequence object (and other *iterables*—a term introduced in [Chapter 4](#) that we'll be more formal about in the next part of this book):

```
>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)           # Make a list from a tuple's items
>>> tmp.sort()              # Sort the list
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> T = tuple(tmp)          # Make a tuple from the list's items
>>> T
('aa', 'bb', 'cc', 'dd')

>>> sorted(T, reverse=True) # Or use the sorted built-in, and save steps
['dd', 'cc', 'bb', 'aa']
```

Here, the `list` and `tuple` built-in functions are used to convert the object to a list and then back to a tuple. Really, both calls make new objects from any sort of iterables, but the net effect is like a conversion.

In some sense, list *comprehensions* can also be used to convert tuples. The following, for example, makes a list from a tuple, adding 20 to each item along the way:

```
>>> T = (1, 2, 3, 4, 5)
>>> L = [x + 20 for x in T] # Like list(T) + expression logic
>>> L
[21, 22, 23, 24, 25]
```

List comprehensions are really *sequence* operations—they always build new lists, but they may be used to iterate over any sequence objects, including tuples, strings, and other lists. As you'll see later in the

book, they even work on some things that are not physically stored sequences—any *iterable* objects will do, including files, which are automatically read line by line. Given this, they may be better called *iteration* tools.

Notice that you’d have to convert the prior example’s list result back to a tuple if your code must care. There is no tuple comprehension in Python (as explored later in this book, parenthesized comprehensions make *generators*), though you simulate one by using `tuple` to force a generator to give up its values:

```
>>> tuple(x + 20 for x in T)      # Tuple "comprehension" = generator + builder
(21, 22, 23, 24, 25)
```

Although tuples don’t have the same *methods* as lists and strings, they do have two of their own—`index` and `count` work as they do for lists, but they are defined for tuple objects:

```
>>> T = (1, 2, 3, 2, 4, 2)      # Tuple methods
>>> T.index(2)                  # Offset of first appearance of 2: index_of_!
1
>>> T.index(2, 2)               # Offset of appearance after offset 2
3
>>> T.count(2)                  # How many 2s are there?
3
```

Also, note that the rule about tuple *immutability* applies only to the top level of the tuple itself, not to its contents. A list inside a tuple, for instance, can be changed as usual:

```
>>> T = (1, [2, 3], 4)
>>> T[1] = 'mod'                # This fails: can't change tuple itself
TypeError: 'tuple' object does not support item assignment

>>> T[1][0] = 'mod'             # This works: can change mutables inside
>>> T
(1, ['mod', 3], 4)
```

For most programs, this one-level-deep immutability is sufficient for common tuple roles. Which, coincidentally, brings us to the next section.

Why Lists and Tuples?

This seems to always be the first question that comes up when teaching beginners about tuples: why do we need tuples if we have lists? Some of the reason is philosophical: a tuple is meant to be a simple association of objects, while a list is intended to be a data structure that changes over time. In fact, this meaning of “tuple” derives from mathematics, as well its frequent use for a row in a relational database table.

The best answer, however, seems to be that the immutability of tuples provides some *integrity*—you can be sure a tuple won’t be changed through another reference elsewhere in a program, but there’s no such guarantee for lists. Tuples and other immutables, therefore, serve a similar role to “constant” declarations in other languages, though the notion of constantness is associated with *objects* in Python, not variables.

Tuples can also be used in places that lists cannot—for example, as dictionary keys (see the sparse matrix example in [Chapter 8](#)). Some built-in operations may also require or imply tuples instead of lists (e.g., the substitution values in the % string formatting expression of [Chapter 7](#)), though some operations have often been generalized to be more flexible. As a rule of thumb, lists are the tool of choice for ordered collections that might need to change; tuples can handle the other cases of fixed associations.

Records Revisited: Named Tuples

In fact, the choice of data types is even richer than the prior section may have implied—Python programmers can choose from an assortment of both built-in core types, and extension types built on top of them. For example, in the prior chapter’s sidebar “[Why You Will Care: List Versus Dictionary Versus Set](#)” on page 204, we saw how to represent record-like information with both a list and a dictionary and noted that dictionaries offer the advantage of more mnemonic keys that label data. As long as we don’t require mutability, tuples can serve similar roles, with positions for record fields like lists:

```
>>> pat = ('Pat', 40.5, ['dev', 'mgr'])           # Tuple record
>>> pat
('Pat', 40.5, ['dev', 'mgr'])

>>> pat[0], pat[2]                               # Access by position
('Pat', ['dev', 'mgr'])
```

As for lists, though, field numbers in tuples generally carry less information than the names of keys in a *dictionary*. To review, here’s the same record recoded as a dictionary with named fields:

```
>>> pat = dict(name='Pat', age=40.5, jobs=['dev', 'mgr']) # Dictionary record
>>> pat['name'], pat['jobs']                             # Access by key
('Pat', ['dev', 'mgr'])
```

In fact, we can convert parts of the dictionary to tuples if needed:

```
>>> tuple(pat.values())                               # Values to tuple
('Pat', 40.5, ['dev', 'mgr'])
>>> list(pat.items())                                 # Items to tuple list
[('name', 'Pat'), ('age', 40.5), ('jobs', ['dev', 'mgr'])]
```

But really, this is a false dichotomy: with extra code, we can implement objects that offer *both* positional and named access to record fields. For example, the `namedtuple` utility, noncore but always available in the standard library’s `collections` module, implements an extension type that adds logic to tuples that allows components to be accessed by both *position* and attribute *name*, and can be converted to dictionary-like form for access by *key* if desired. Attribute names come from classes and are not exactly dictionary keys, but they are similarly mnemonic:

```
>>> from collections import namedtuple               # Import extension type
>>> Rec = namedtuple('Rec', ['name', 'age', 'jobs']) # Make a generated class
>>> pat = Rec('Pat', age=40.5, jobs=['dev', 'mgr']) # A named-tuple record
>>> pat
Rec(name='Pat', age=40.5, jobs=['dev', 'mgr'])

>>> pat[0], pat[2]                                   # Access by position
('Pat', ['dev', 'mgr'])
>>> pat.name, pat.jobs                               # Access by attribute
('Pat', ['dev', 'mgr'])
```

Converting to a dictionary also supports key-based behavior when needed:

```
>>> D = pat._asdict()                               # Dictionary-like form
>>> D['name'], D['jobs']                             # Access by key too
(('Pat', ['dev', 'mgr'])
>>> D
{'name': 'Pat', 'age': 40.5, 'jobs': ['dev', 'mgr']}
```

As you can see, named tuples are a tuple/class/dictionary *hybrid*. They also represent a classic *trade-off*. In exchange for their extra utility, they require extra code to use (the two startup lines in the preceding examples that import the type and make the class) and incur some performance costs to work this magic. Still, they are an example of the kind of custom data types that we can build on top of built-in

types like tuples when extra utility is desired. They are also *extensions*, not core types—they live in the standard library and fall into the same category as [Chapter 5](#)’s `Fraction` and `Decimal`—so we’ll delegate to the Python library manual for more details.

Watch for a final rehash of this record representation thread when we explore how user-defined *classes* compare in [Chapter 27](#). As you’ll find there, classes label fields with names too, but can also provide program *logic* to process the record’s data in the same code package.

Files

You may already be familiar with the notion of files, which are named storage compartments on your PC, phone, or other computer that are managed by your operating system. The last major built-in object type that we’ll examine on our object-types tour provides a way to access those files inside Python programs.

In short, the built-in `open` function creates a Python file object, which serves as a link to a file residing on your device. After calling `open`, you can transfer strings of data to and from the associated external file by calling the returned file object’s methods.

Compared to the types you’ve seen so far, file objects are outliers. They are considered a core type because they are created by a built-in function, but they’re not numbers, sequences, or mappings, and they don’t respond to expression operators; they export only *methods* for common file-processing tasks. Most file methods are concerned with performing input from and output to the external file associated with a file object, but other file methods allow us to seek to a new position in the file, flush output buffers, and so on. [Table 9-2](#) summarizes common file operations.

Table 9-2. Common file operations

Operation	Interpretation
<code>output = open(r'C:\data', 'w')</code>	Create output file (Windows path, 'w' = write)
<code>input = open('/home/me/data', 'r')</code>	Create input file (Unix path, 'r' = read)
<code>input = open('data')</code>	Create input file (current directory, 'r' is default)
<code>aString = input.read()</code>	Read entire file into a single string
<code>aString = input.read(N)</code>	Read up to next <i>N</i> characters (or bytes) into a string
<code>aString = input.readline()</code>	Read next line (including <code>\n</code> newline) into a string
<code>aList = input.readlines()</code>	Read entire file into a list of line strings (with <code>\n</code>)
<code>output.write(aString)</code>	Write a string of characters (or bytes) into a file
<code>output.writelines(aList)</code>	Write all line strings in a list into a file (verbatim)
<code>output.close()</code>	Manual close (done for you when file is collected)
<code>output.flush()</code>	Flush output buffer to disk without closing
<code>anyFile.seek(N)</code>	Change file position to offset <i>N</i> for next operation
<code>for line in open('data'): use line</code>	File iterators read line by line
<code>open('f.txt', encoding='utf-8')</code>	Unicode text files (using <code>str</code> strings)
<code>open('f.bin', 'rb')</code>	Bytes files (using <code>bytes</code> strings)
<code>codecs.open('f.txt',...)</code>	Alternative Unicode text-file interface

Opening Files

To open a file, a program calls the built-in `open` function, with the external filename first, followed by a processing mode. Both arguments are strings. The call returns a file *object*, which in turn has *methods* for data transfer:

```
afile = open(filename, mode)
afile.method()
```

The first argument to `open`, the external *filename*, may include a platform-specific and *absolute* (complete) or *relative* (partial) directory-path prefix that identifies a file's location in the host device's filesystem. The filesystem is just a hierarchy of folders that stores files and nested folders. A filename `a/b/file.txt`, for example, includes the path prefix `a/b` that leads to a file on Unix (e.g., macOS, Linux, or Android), and `a\b\file.txt` does the same on Windows.

If *filename* has *no* directory-path prefix at all, the file is assumed to exist in, and hence is *relative* to, the current working directory (*CWD*). The *CWD* is the folder from which a script is run (e.g., where you are in a console when you launch a Python command). In a REPL, the *CWD* is wherever you are working at the time; to see what the *CWD* is, run a `pwd` in most system shells, or Python's `os.getcwd()` after `import os` in a REPL.

As you'll see in [Chapter 37](#)'s expanded file coverage, the *filename* may also contain non-ASCII *Unicode* characters that Python automatically translates to and from the underlying host's encoding. These characters may be provided in the filename literally, or in a pre-encoded byte string that you'll learn about later in this book.

The second argument to `open`, processing *mode*, is typically the string `'r'` to open for text input (the default), `'w'` to create and open for text output, or `'a'` to open for appending text to the end (e.g., for adding to logfiles). The processing mode argument can specify additional options:

- Adding a `b` to the mode string (e.g., `'wb'`) allows for processing *binary* file content. End-of-line translations and Unicode encodings used for text are turned off.
- Adding a `+` to the mode (e.g., `'r+'`) opens the file for *both* input and output. You can read and write to the same file object, often in conjunction with `seek` operations to reposition in the file.

Both of the first two arguments to `open` must be Python strings. An optional third argument takes an integer to control output *buffering*—passing a zero means that output is unbuffered (it's transferred to the external file immediately on a write method call), and additional arguments may be provided for special types of files (e.g., the string name of a Unicode *encoding* for text files). You can also use *name=value* keywords to pass open arguments (e.g., `file=name, mode=mode`), though this is somewhat above our pay grade in this chapter.

We'll cover file fundamentals and explore some basic examples here, but we won't go into all file-processing mode options; run `help(open)` in a REPL or consult the Python library manual for additional details.

Using Files

Once you make a file object with `open`, you can call its methods to read from or write to the associated external file. In all cases, file content takes the form of strings in Python programs; reading a file returns its content in strings, and content is passed to the write methods as strings. Reading and writing

methods come in multiple flavors; [Table 9-2](#) lists the most common. Here are a few points of orientation up front:

File iterators may be best for reading text lines

Though the reading and writing methods in the table are common, keep in mind that probably the best way to read lines from a text file today is to not read the file at all—as you’ll see in [Chapter 14](#), files also have an *iterator* that automatically reads one line at a time in a `for` loop, list comprehension, or other iteration context.

Content is strings, not objects

Notice in [Table 9-2](#) that content *read* from a file always comes back to your script as a string, so you’ll have to convert it to a different type of object if a string is not what you need. Similarly, file *write* operations do not add any sort of formatting and do not convert objects to strings automatically, so you must convert if needed and format as desired. Because of this, the tools we have already met to convert objects from and to strings (e.g., `int`, `float`, `str`, and string formatting) come in handy when dealing with files. Also note that newlines, added by `print` but not file *writes*, may have to be skipped if text from file *reads* is sent to `print` to avoid double spacing.

Python also includes advanced standard-library tools for handling generic object storage (the `pickle` module), for dealing with packed binary data in files (the `struct` module), and for processing special types of content such as JSON and CSV text. We’ll demo these later in this chapter, but Python’s manuals document them in full.

Files are buffered and seekable

By default, output files are always *buffered*, which means that text you write may not be transferred from memory to disk immediately—closing a file, or running its `flush` method, forces the buffered data to disk. You can avoid buffering with extra `open` arguments, but it may impede performance. Python files are also *random-access* on a byte-offset basis—their `seek` method allows your scripts to jump around to read and write at specific locations.

close may be optional: auto-close on collection

Calling the file `close` method terminates your connection to the external file, releases its system resources, and flushes its buffered output to disk if any is still in memory. As discussed in [Chapter 6](#), an object’s memory space is automatically reclaimed as soon as the object is no longer referenced anywhere in the program. When *file* objects are reclaimed, Python also automatically *closes* them if they are still open (this also happens to open files when a program shuts down). This means you don’t always need to manually close your files in Python, especially those in simple scripts with short runtimes, and temporary files used by a single line or expression.

On the other hand, manual `close` calls don’t hurt and are a good habit to form, especially in long-running systems and code run at the REPL. Strictly speaking, this auto-close of files is an implementation artifact of the standard *CPython*, and not part of the language definition—it may change over time, may not happen when you expect it to in interactive REPLs, and may not work the same in Python implementations whose garbage collectors reclaim space differently than CPython. In fact, when many files are opened within loops, some Pythons may *require* `close` calls to free up system resources immediately, before garbage collection can get around to freeing objects. `Close` calls may sometimes also be required to flush buffered output of file objects not yet reclaimed. For an alternative and automatic way to ensure closes, watch for the file object’s *context manager* ahead.

Files in Action

Let's work through an example that demonstrates file-processing basics. The following code begins by opening a new text file for output, writing two lines (strings terminated with a newline marker, `\n`), and closing the file. Later, the example opens the same file again in input mode and reads the lines back one at a time with `readline`:

```
>>> myfile = open('myfile.txt', 'w')      # Open for text output: create/empty
>>> myfile.write('hello text file\n')     # Write a line of text: string
16
>>> myfile.write('goodbye text file\n')
18
>>> myfile.close()                       # Ensure output is flushed to disk

>>> myfile = open('myfile.txt')           # Open for text input: 'r' is default
>>> myfile.readline()                    # Read the lines back
'hello text file\n'
>>> myfile.readline()
'goodbye text file\n'
>>> myfile.readline()                    # Empty string: end-of-file
''
```

Notice that the third `readline` call returns an *empty string*—this is how most file read methods tell you that you've reached the *end of the file* (and as you'll see ahead, the empty string is inherently false in logical tests). Empty lines in the file instead come back as strings containing just a newline character (`'\n'`), not as empty strings.

Also notice that file write calls return the number of characters written; this is normally superfluous apart from error checks but is echoed in a REPL like this. This example writes each line of text, including its newline terminator, `\n`, as a string. Write methods don't add the newline character for us, so we must include it to properly terminate our lines; without this, the next write will simply extend the current line in the file.

If you want to display the file's content with newline characters interpreted, read the entire file into a string *all at once* with the file object's `read` method and print it (stringing together the open and read like this runs left to right and doesn't allow for an explicit `close`, but it doesn't matter for simple input at the REPL):

```
>>> open('myfile.txt').read()           # Read all at once into string
'hello text file\ngoodbye text file\n'

>>> print(open('myfile.txt').read())     # User-friendly display
hello text file
goodbye text file
```

And if you want to scan a text file line by line, *file iterators* are often your best option:

```
>>> for line in open('myfile.txt'):      # Use file iterators, not reads
...     print(line, end='')              # Don't add another \n: line has one!
...
hello text file
goodbye text file
```

When coded this way, the temporary file object created by `open` will automatically read and return one line on each loop iteration. This form is usually easiest to code, light on memory use, and may be faster than some other options (depending on many variables, of course). Since we haven't reached statements or iterators yet, though, you'll have to wait until [Chapter 14](#) for a more complete explanation of this code.

As noted, files content is always strings, so other kinds of objects must be converted for write. The `str` call and concatenating separators and newlines suffice and emulate what `print` does automatically (but if you like the sugarcoating provided by `print`, stay tuned for its coverage in the next part of this book, where you'll learn how to route its display to a file you make first with `open`):

```
>>> myfile = open('myfile2.txt', 'w')      # Nonstring objects fail
>>> myfile.write(3.14)
TypeError: write() argument must be str, not float

>>> myfile.write(str(3.14) + '\n')          # Convert (and emulate print)
5
>>> myfile.close()
>>> open('myfile2.txt').read()              # Can reconvert with float()
'3.14\n'
```

Incidentally, the files we've made here show up in the *CWD*, because we didn't provide a path prefix in their filenames. In Python, you can check what the *CWD* is and get a listing of the files there, with the `os` standard-library module:

```
>>> import os
>>> os.getcwd()                             # Show the current working directory
/Users/me/code/Chapter09
>>> os.listdir()                            # List files here (or in a passed path)
['myfile2.txt', 'myfile.txt']
```

This was run on macOS and mirrors shell commands, but such tools are useful in many programs. The takeaway is that filename *myfile.txt* in this *CWD* is equivalent to */Users/me/code/Chapter09/myfile.txt* in `open` and other tools.



Coding Windows paths: If you opt to provide full directory paths for files on Windows, they may require special handling because the Windows `\` path separator is also used for string escapes in Python per [Chapter 7](#). As noted there, `open` accepts Unix-style forward slashes in place of backward slashes on Windows, so any of the following forms work for directory paths on Windows:

```
open(r'C:\Users\me\code\newdata.txt')      # Raw strings
open('C:/Users/me/code/newdata.txt')       # Forward slashes
open('C:\\Users\\me\\code\\newdata.txt')    # Doubled-up escapes
```

The raw string form in the first command is useful to turn off unintended escapes (e.g., `\n`), though the other two options make the escapes issue moot. On Unix, you'll simply use forward slashes, of course (and drop the Windows drive letter: your drives are mounted, not segregated).

Text and Binary Files: The Short Story

Strictly speaking, the examples in the prior section use *text* files. More generally, file type is determined by the second argument to `open`, the mode string—including a “*b*” in it means *binary*, which is sharply distinguished from *text*:

- *Text files* represent content as a normal `str` string, perform Unicode encoding and decoding automatically, and perform newline translation by default. This mode is useful for processing text of all kinds.

- *Binary files* represent content as a special bytes string and allow programs to access file content unaltered. This mode is useful for processing nontext content like media.

Programs that deal only with simple text like ASCII can get by with the basic text-file interface used in the prior examples, and normal strings. All text strings are technically Unicode in Python, but ASCII users will not generally notice because it's a subset of Unicode (every ASCII file is a Unicode file, even if its character range is limited).

If you need to handle non-ASCII text or byte-oriented data, though, you'll need to match object types to file modes—bytes strings for binary files, and normal str strings for text files. Because text files implement Unicode encodings, you also should not open a binary data file in text mode: decoding its content to Unicode text will likely fail.

Let's turn to a brief example. When you write and read a *binary* file, you send and receive a bytes object—a sequence of small integers that represent absolute byte values (which may or may not correspond to characters), and which is coded with a leading b but looks and feels almost exactly like a normal text string:

```
>>> myfile = open('myfile3.bin', 'wb')           # Make binary file: wb=write binary
>>> myfile.write(b'\x00\x01hack\x02\x03').      # Bytes string holds binary data
8
>>> myfile.close()

>>> data = open('myfile3.bin', 'rb').read()      # Read binary file: rb=read binary
>>> data                                          # Raw, unaltered bytes returned
b'\x00\x01hack\x02\x03'
>>> data[2:6]                                  # Bytes act like text strings
b'hack'
>>> byte = data[2:6][0]                         # But really small 8-bit integers
>>> byte, chr(byte), bin(byte)
(104, 'h', '0b1101000')
```

In addition, binary files do not perform any *newline* translation on content, but text files by default map all forms to and from \n when read and written. Text files also implement Unicode *encodings* on transfers, using an optional encoding name passed to the encoding argument of open. If encoding is not passed (as in earlier examples), files fall back on the underlying platform's default, which may not be interoperable with other hosts or files.

Per the start of this chapter, though, that's as much as we're going to say about Unicode text and binary data files here, and just enough to understand upcoming examples in this chapter. If you're anxious to dive into this topic further, see either the preview in [Chapter 4](#) or wait for the full story in [Chapter 37](#).

For this chapter, let's move on to a handful of more substantial file examples that demonstrate common ways to store Python object values in files.

Storing Objects with Conversions

Our next example writes a variety of Python objects to a *text* file on multiple lines. We wrote a single number to a file earlier, but are kicking it up a notch here to demo more conversions. Again, file content is strings in our code, and write methods do not do any to-string formatting (for space, this chapter omits write return values from here on):

```

>>> X, Y, Z = 62, 63, 64                                # Native Python objects
>>> S = 'Text'                                           # Must be strings to store in file
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]

>>> F = open('datafile.txt', 'w')                        # Create output text file
>>> F.write(S + '\n')                                    # Terminate lines with \n
>>> F.write(f'{X},{Y},{Z}\n')                            # Convert numbers to strings
>>> F.write(str(L) + '$' + str(D) + '\n')                # Convert and separate with $
>>> F.close()

```

Once we have created our file, we can inspect its contents by opening it and reading it into a string (strung together as a single operation here). Notice that the interactive echo gives the exact character contents, while the `print` operation interprets embedded newline characters to render a more user-friendly display:

```

>>> chars = open('datafile.txt').read()                  # Raw string display
>>> chars
"Text\n62,63,64\n[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> print(chars)                                         # User-friendly display
Text
62,63,64
[1, 2, 3]${'a': 1, 'b': 2}

```

We now have to use other conversion tools to translate from the strings in the text file to real Python objects. As Python never converts strings to numbers (or other types of objects) automatically, this is required if we need to gain access to normal object tools like indexing, addition, and so on:

```

>>> F = open('datafile.txt')                             # Open again
>>> line = F.readline()                                   # Read one line
>>> line
'Text\n'
>>> line.rstrip()                                         # Remove newline
'Text'

```

For this first line, we used the string `rstrip` method to get rid of the trailing newline character; a `line[:-1]` slice would work, too, but only if we can be sure all lines end in the `\n` character (the last line in a file sometimes does not).

So far, we've read the line containing the string. Now let's grab the next line, which contains numbers, and parse out (that is, extract) the objects on that line:

```

>>> line = F.readline()                                  # Next line from file
>>> line                                                  # It's a string here
'62,63,64\n'
>>> parts = line.rstrip().split(',')                     # Split (parse) on commas
>>> parts
['62', '63', '64']

```

We used the string `split` method here to chop up the line on its comma delimiters (after removing the trailing `\n` with `rstrip` as before—its result is a new string on which we run `split`). The result is a list of substrings containing the individual numbers. We still must convert from strings to integers, though, if we wish to perform math on these:

```

>>> int(parts[1])                                         # Convert from string to int
63
>>> numbers = [int(P) for P in parts]                     # Convert all in list at once
>>> numbers
[62, 63, 64]

```

As we have learned, `int` translates a string of digits into an integer object, and the list comprehension expression introduced in Chapters 4 and 8 can apply the call to each item in our list all at once (again, you'll find more on list comprehensions later in this book). Nit: we didn't have to use `rstrip` to delete the `\n` at the end of the line, because `int` and some other converters quietly ignore whitespace around digits; still, being explicit is often best.

Finally, to convert the stored list and dictionary in the third line of the file, we can run them through `eval`, a built-in function we first met in Chapter 5, that treats a string as a piece of executable program code (technically, a string containing a Python expression, with trade-offs discussed in the next section):

```
>>> line = F.readline()           # Next line from file
>>> line
"[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> parts = line.split('$')       # Split (parse) on $
>>> parts
['[1, 2, 3]', '{"a': 1, 'b': 2}\n"]
>>> eval(parts[0])                # Convert to any object type
[1, 2, 3]
>>> objects = [eval(P) for P in parts] # Do same for all in list
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]
```

Because the end result of all this parsing and converting is a list of normal Python objects instead of strings, we can now apply list and dictionary operations to them in our script.

Storing Objects with pickle

Using `eval` to convert from strings to objects, as demonstrated in the preceding code, is a powerful tool. In fact, sometimes it's *too* powerful. `eval` will happily run any Python expression—even one that might delete all the files on your computer, given the necessary permissions. If you really want to store native Python objects, but you don't want to run file content as program code, Python's standard-library `pickle` module can help.

The `pickle` module is a more advanced tool that allows us to store almost any Python object in a file directly, with no to- or from-string conversion requirement on our part. It's like a super-general data formatting and parsing utility. To store a dictionary in a file, for instance, we pickle it directly after using binary mode to open the file:

```
>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'wb')
>>> import pickle
>>> pickle.dump(D, F)              # Pickle almost any object to file
>>> F.close()
```

Then, to get the dictionary back later, we simply use `pickle` again to re-create it, again using a binary-mode file:

```
>>> F = open('datafile.pkl', 'rb')
>>> E = pickle.load(F)             # Load almost any object from file
>>> E
{'a': 1, 'b': 2}
```

We get back an equivalent dictionary object, with no manual splitting or converting required. The `pickle` module performs what is known as *object serialization*—converting objects to and from strings of bytes—but requires very little work on our part. In fact, `pickle` internally translates our dictionary to

a bytes-string form; it's not much to look at, and may vary for protocol options and future morph, but mandates binary-mode files:

```
>>> open('datafile.pkl', 'rb').read()          # Format is prone to change!
b'\x80\x04\x95\x11\x00\x00\x00\x00 ...etc... \x8c\x01a\x94K\x01\x8c\x01b\x94K\x02u.'
```

Because pickle can reconstruct the object from this format, we don't have to deal with it ourselves. For more on the pickle module, see the Python standard-library manual, or import pickle and pass it to help interactively. While you're exploring, also take a look at the shelve module. shelve is a tool that uses pickle to store Python objects in an access-by-key filesystem, which is beyond our scope here (though you will get to see an example of shelve in action in [Chapter 28](#), and other pickle examples in [Chapters 31](#) and [37](#)).

Storing Objects with JSON

The prior section's pickle module translates nearly arbitrary Python objects to a proprietary format developed specifically for Python, and honed for performance over many years. JSON is a newer data interchange format, which is both programming-language-neutral and supported by a variety of systems. *MongoDB*, for instance, stores data in a JSON document database (using a binary JSON format), and JSON is common in configuration roles.

JSON does not support as broad a range of Python object types as pickle, but its portability is an advantage in some contexts, and it represents another way to serialize a specific category of Python objects for storage and transmission. Moreover, because JSON is so close to Python dictionaries and lists in syntax, the translation to and from Python objects is trivial, and is automated by the json standard-library module.

For example, a Python dictionary with nested structures is very similar to JSON data, though Python's variables and expressions support richer structuring options (any part of the following can be an arbitrary expression in Python):

```
>>> who = dict(first='Pat', last='Smith')
>>> rec = dict(name=who, job=['dev', 'mgr'], age=40.5)
>>> rec
{'name': {'first': 'Pat', 'last': 'Smith'}, 'job': ['dev', 'mgr'], 'age': 40.5}
```

The final dictionary format displayed here is a valid literal in Python code, and almost passes for JSON when printed as is, but the json module makes the translation official—here translating Python objects to and from a JSON serialized string representation in memory:

```
>>> import json
>>> json.dumps(rec)
'{"name": {"first": "Pat", "last": "Smith"}, "job": ["dev", "mgr"], "age": 40.5}'

>>> S = json.dumps(rec)          # Python => JSON
>>> O = json.loads(S)            # JSON => Python
>>> O
{'name': {'first': 'Pat', 'last': 'Smith'}, 'job': ['dev', 'mgr'], 'age': 40.5}
>>> O == rec
True
```

It's similarly straightforward to translate Python objects to and from JSON data strings in files. Prior to being stored in a file, your data is simply Python objects; the JSON module re-creates them from the JSON textual representation when it loads it from the file. In both cases, we use text files, because JSON is text:

```

>>> file = open('testjson.txt', 'w')
>>> json.dump(rec, fp=file, indent=4)      # Python => JSON
>>> file.close()
>>> print(open('testjson.txt').read())     # Human-readable file content
{
    "name": {
        "first": "Pat",
        "last": "Smith"
    },
    "job": [
        "dev",
        "mgr"
    ],
    "age": 40.5
}
>>> P = json.load(open('testjson.txt'))    # JSON => Python
>>> P
{'name': {'first': 'Pat', 'last': 'Smith'}, 'job': ['dev', 'mgr'], 'age': 40.5}
>>> P == rec
True

```

Once you’ve translated from JSON text, you process the data using normal Python object operations in your script. For more details on JSON-related topics, see Python’s library manuals and the web at large. You could, of course, store real Python dictionaries and lists in an imported Python *.py* module file, and the data would be just as readable and editable; JSON, however, is not run as code, and may be perceived by some as more interoperable today.

Note that strings are all *Unicode* in JSON to support richer forms of text. Since Python strings in memory simply *are* Unicode, the distinction matters most when transferring text to and from files. You’ll learn how to apply Unicode encodings to JSON (and other) files and data in [Chapter 37](#).

Storing Objects with Other Tools

For other common ways to deal with formatted data files, see the standard library’s `struct` and `csv` modules. Very briefly, the `struct` module can both create and parse packed binary data of the sort often shared with C programs:

```

>>> import struct
>>> data = struct.pack('i6s', 62, b'Python')    # Pack an int and str in bytes
>>> data
b'>\x00\x00\x00Python'

>>> file = open('data.bin', 'wb')                # Write/read file, and unpack
>>> file.write(data)                             # Binary data in binary files
>>> file.close()

>>> struct.unpack('i6s', open('data.bin', 'rb').read())
(62, b'Python')

```

And the `csv` module parses and creates comma-separated value (CSV) data in files and strings; it doesn’t map as directly to Python objects (and requires post-parse conversions), but is another way to map value to and from files:

```

>>> import csv
>>> rdr = csv.reader(open('csvdata.txt'))
>>> for row in rdr: print(row)
...
['a', 'bbb', 'cc', 'dddd']
['11', '22', '33', '44']

```


For additional data storage ideas like YAML and SQLite, see the overview of database tools in [Chapter 1](#).

File Context Managers

You'll also want to watch for [Chapter 34](#)'s in-depth discussion of the file's context manager support. Though more a feature of exception processing than files themselves, it allows us to wrap file-processing code in a logic layer that ensures that the file will be closed (and if needed, have its output flushed to disk) automatically on statement exit, instead of relying on the auto-close during garbage collection or manual close calls. As a preview:

```
with open('data.txt') as myfile:           # File closed on "with" exit
    for line in myfile:                     # See Chapter 34 for details
        ...use line here...
```

The `with` statement closes the temporary file on exit, whether an error occurs or not. The `try/finally` statement that we'll also study in [Chapter 34](#) can provide similar functionality, but at some cost in extra code—three extra lines, to be precise (though we can often avoid both options and let Python close files for us automatically):

```
myfile = open('data.txt')
try:                                     # General termination handler
    for line in myfile:                 # See Chapter 34 for details
        ...use line here...
finally:
    myfile.close()
```

The `with` context manager scheme ensures release of system resources in all Pythons and may be more useful for output files to guarantee buffer flushes; unlike the more general `try`, though, it is also limited to objects that support its protocol. Since both these options require more information than we have yet obtained, however, we'll postpone the rest of their stories until later in this book.

Other File Tools

There are additional, more specialized file methods shown in [Table 9-2](#), and even more that are not in the table. For instance, as mentioned earlier, `seek` resets your current position in a file (the next read or write happens at that position); `flush` forces buffered output to be written out to disk without closing the connection (by default, files are always buffered); and `readlines` and `writelines` process file content in line lists.

The Python standard-library manual and other reference resources provide complete details on file methods, but for a quick look, run a `dir` or `help` call interactively, passing in `open` or a file object made with it. And for more file-processing examples, watch for [Chapter 37](#)'s extended coverage, as well as the sidebar [“Why You Will Care: File Scanners” on page 333](#), which sketches common file-scanning patterns with statements we have not yet covered here.

Also, note that although the `open` function and the file objects it returns are your main interface to external files in a Python script, there are additional file-related tools in the Python toolset. Prominent among these are:

Standard streams

Preopened file objects in the `sys` module, such as `sys.stdout`, connected by default to the UI where a script is run (see [“Print Operations” on page 282](#) for details)

Descriptor files in the os module

Integer file handles that support lower-level tools such as read-only access (see also the “x” mode modifier in `open` for exclusive creation)

Sockets, pipes, and FIFOs

File-like objects used to synchronize processes or communicate over networks

Access-by-key files known as shelves

Used to store unaltered and pickled Python objects directly, by key (see [Chapter 28](#) for an example)

Shell-command streams

Tools such as `os.popen` and `subprocess.Popen` that support spawning shell commands and reading and writing to their standard streams (see [Chapter 21](#) for an `os.popen` example)

The third-party open source domain offers even more file-like tools, including support for communicating with serial ports in the *PySerial* extension and interactive programs in the *pexpect* system. Consult the web at large for additional information on file-like tools.

For code spelunkers, it’s also worth noting that Python’s `open` function is really just an interface to tools in its standard-library `io` module, which adds logic on top of the underlying system’s file tools to make them portable and efficient. If you’re looking for docs, implementation details, or customization hooks, look for this module in all the usual places. `io` is really a folder called a module package—a structure for larger code that we’ll study later.

Core Types Review and Summary

Now that we’ve seen all of Python’s built-in objects in action, let’s wrap up our object-types tour by reviewing some of the properties they share. [Table 9-3](#) classifies all the major types we’ve studied so far according to the type categories introduced earlier. Here are some points to remember:

- Objects share operations according to their category. For instance, *sequence* objects—strings, lists, and tuples—all share sequence operations such as concatenation, length, and indexing.
- Only *mutable* objects—lists, dictionaries, and sets—may be changed in place. You cannot change numbers, strings, or tuples in place, but can make a new one with a different value.
- Files export only *methods*, so mutability doesn’t really apply to them—their state may be changed when they are processed, but this isn’t quite the same as Python object mutability.
- “Numbers” in [Table 9-3](#) includes all number types: integer, floating point, complex, decimal, and fraction.
- “Strings” in [Table 9-3](#) includes all string types: `str` for text, as well as `bytes` for binary data. Exception: the `bytearray` string type convolutes categorization because it is a mutable string.
- Sets are something like the keys of a valueless dictionary, but they don’t map to values and are not ordered, so sets are neither a mapping nor a sequence type. Exception: `frozenset` is an immutable variant of `set`.
- In addition to type category operations, all the objects types in [Table 9-3](#) have callable methods in Python today, which are generally specific to their type.

Table 9-3. Object classifications

Object type	Category	Mutable?
Numbers	Numeric	No
Strings	Sequence	No
Lists	Sequence	Yes
Dictionaries	Mapping	Yes
Tuples	Sequence	No
Files	Extension	N/A
Sets	Set	Yes
Frozenset	Set	No
bytearray	Sequence	Yes

Why You Will Care: Operator Overloading

In **Part VI** of this book, you'll learn that objects implemented with classes can pick and choose from these categories arbitrarily. For instance, if we want to provide a new kind of specialized sequence object that is consistent with built-in sequences, we can code a class that overloads things like indexing and concatenation:

```
class MySequence:
    def __getitem__(self, index):
        # Called on self[index], others
    def __add__(self, other):
        # Called on self + other
    def __iter__(self):
        # Preferred in iterations
```

and so on. We can also make the new object mutable or not by selectively implementing methods called for in-place change operations (e.g., `__setitem__` is called on `self[index]=value` assignments). Although it's beyond this book's scope, it's also possible to implement new objects in an external language like C as extension types. For these, we fill in C function pointer slots to choose between number, sequence, and mapping operation sets, and similarly choose immutability constraints.

Object Flexibility

This part of the book introduced a number of *compound* object types—collections with components. In general:

- Lists, dictionaries, and tuples can hold any kind of object.
- Sets can contain any type of immutable object.
- Lists, dictionaries, and tuples can be arbitrarily nested.
- Lists, dictionaries, and sets can dynamically grow and shrink.

Because they support arbitrary structures, Python's compound object types are good at representing complex information in programs. For example, values in dictionaries may be lists, which may contain

tuples, which may contain dictionaries, and so on. The nesting can be as deep as needed to model the data to be processed.

In code, the following interaction defines a tree of nested compound sequence objects, sketched in [Figure 9-1](#). To access its components, you may include as many index operations as required. Python evaluates the indexes from left to right and fetches a reference to a more deeply nested object at each step. [Figure 9-1](#) may seem a pathologically complicated data structure, but it illustrates the syntax used to access nested objects in general:

```
>>> L = ['abc', [(1, 2), ([3], 4)], 5]
>>> L[1]
[(1, 2), ([3], 4)]
>>> L[1][1]
([3], 4)
>>> L[1][1][0]
[3]
>>> L[1][1][0][0]
3
```

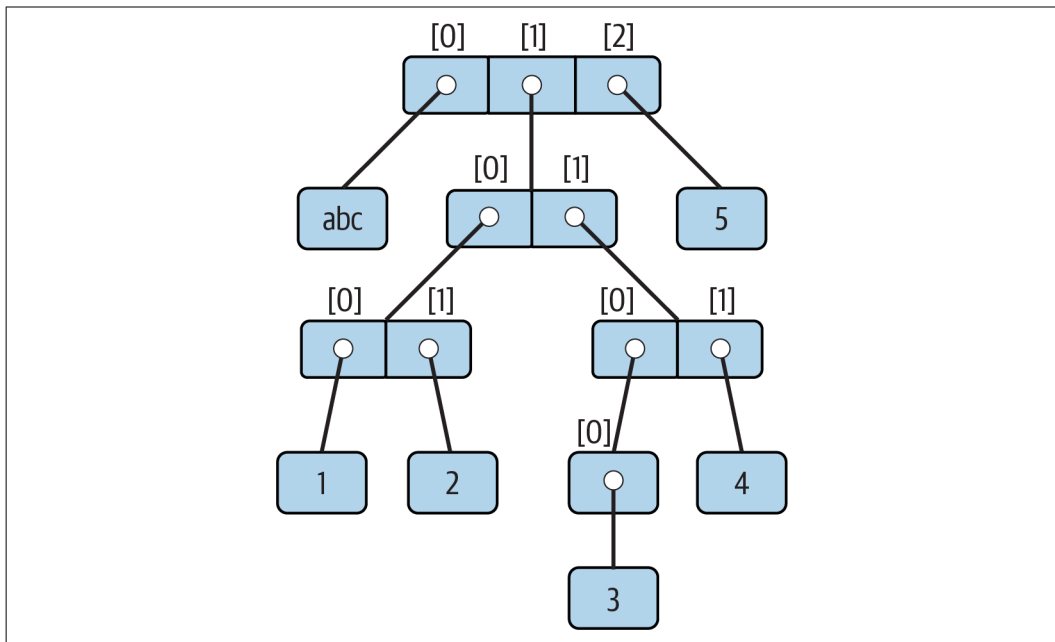


Figure 9-1. A nested object tree with the offsets of its components

References Versus Copies

[Chapter 6](#) mentioned that assignments always store references to objects, not copies of those objects. In practice, this is usually what you want. Because assignments can generate multiple references to the same object, though, it's important to be aware that changing a mutable object in place may affect other references to the same object elsewhere in your program. If you don't want such behavior, you'll need to tell Python to copy the object explicitly.

We studied this phenomenon in [Chapter 6](#), but it can become more subtle when larger objects of the sort we've explored since then come into play. For instance, the following example creates a list assigned

to X, and another list assigned to L that embeds a reference back to list X. It also creates a dictionary D that contains another reference back to list X:

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']           # Embed references to X's object in two others
>>> D = {'x':X, 'y':2}
```

At this point, there are three references to the first list created: from the name X, from inside the list assigned to L, and from inside the dictionary assigned to D. The situation is illustrated in [Figure 9-2](#).

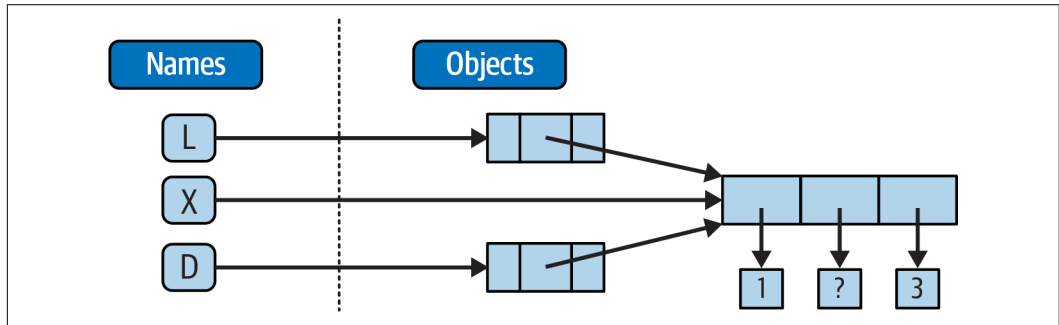


Figure 9-2. Shared objects: changing from X makes it look different from L and D too

Because lists are mutable, changing the shared list object from any of the three references also changes what the other two reference:

```
>>> X[1] = 'surprise'           # Changes all three references!
>>> L
['a', [1, 'surprise', 3], 'b']
>>> D
{'x': [1, 'surprise', 3], 'y': 2}
```

References are a higher-level analogue of pointers in other languages that are always followed when used. Although you can't grab hold of the reference itself, it's possible to store the same reference in more than one place (variables, lists, and so on). This is a feature—you can pass a large object around a program without generating expensive copies of it along the way. If you really want to avoid the potential side effects of shared references, however, you can request *copies*, in one of a number of ways:

- Slice expressions with empty limits (`L[:]`) copy sequences.
- The dictionary, set, and list copy method (`X.copy()`) copies a dictionary, set, or list.
- Some built-in functions, such as `list` and `dict` make copies (`list(L)`, `dict(D)`, `set(S)`).
- The copy standard-library module makes full (“recursive”) copies when needed.

For example, say you have a list and a dictionary, and you don't want their values to be changed through other variables:

```
>>> L = [1,2,3]
>>> D = {'a':1, 'b':2}
```

To prevent this, simply assign copies to the other variables, not references to the same objects:

```
>>> A = L[:]                    # Instead of A = L (or list(L), L.copy())
>>> B = D.copy()               # Instead of B = D (ditto for sets)
```

This way, changes made from the other variables will change the copies, not the originals:

```
>>> A[1] = 'Py'
>>> B['c'] = 'code'           # Changes copies, not originals
>>>
>>> L, D
([1, 2, 3], {'a': 1, 'b': 2})
>>> A, B
([1, 'Py', 3], {'a': 1, 'b': 2, 'c': 'code'})
```

In terms of our original example, you can avoid the reference side effects by slicing the original list instead of simply naming it:

```
>>> X = [1, 2, 3]
>>> L = ['a', X[:], 'b']      # Embed copies of X's object
>>> D = {'x':X[:], 'y':2}
```

This changes the picture in [Figure 9-2](#)—L and D will now point to *different* lists than X. The net effect is that changes made through X will impact only X, not L and D; similarly, changes to L or D will not impact X.

One final note on copies: empty-limit slices and the dictionary copy method only make *top-level* copies; that is, they do not copy nested data structures, if any are present. If you need a complete, fully independent copy of a deeply nested data structure (like the various record structures we’ve coded in recent chapters), use the standard copy module, introduced in [Chapter 6](#):

```
import copy
X = copy.deepcopy(Y)          # Fully copy an arbitrarily nested object Y
```

This call traverses objects to copy all their parts, no matter how deep they may be. This is a much rarer case, though, which is why you have to say more to use this scheme. References are usually what you will want; when they are not, slices and copy methods are usually as much copying as you’ll need to do.

Comparisons, Equality, and Truth

All Python objects also respond to comparisons: tests for equality, relative magnitude, and so on. We’ve seen comparison at work on specific objects in earlier chapters but can finally summarize the general rules.

In short, Python comparisons always inspect all parts of compound objects until a result can be determined. When nested objects are present, Python automatically traverses data structures to apply comparisons from left to right, and as deeply as needed. The first difference found along the way determines the comparison result.

This is sometimes called a *recursive* comparison—the same comparison requested on the top-level objects is applied to each of the nested objects, and to each of *their* nested objects, and so on, until a result is found. Later in this book ([Chapter 19](#)) you’ll learn how to write recursive functions of your own that work similarly on nested structures. For now, think about comparing all the linked pages at two websites if you want a metaphor for such structures, and a reason for writing recursive functions to process them.

In terms of core objects, the recursion is automatic. For instance, a comparison of list objects compares all their components automatically until a mismatch is found or the end is reached:

```
>>> L1 = [1, ('a', 3)]        # Same value, but different objects
>>> L2 = [1, ('a', 3)]
>>> L1 == L2, L1 is L2       # Same value? Same object?
(True, False)
```

Here, L1 and L2 are assigned lists that are equivalent but distinct objects. As a review of [Chapter 6](#)'s coverage, because of the nature of Python references, there are two ways to test for equality:

- **The == operator tests value equivalence.** Python performs an equivalence test, comparing all nested objects recursively.
- **The is operator tests object identity.** Python tests whether the two are really the same object (i.e., live at the same address in memory).

In the preceding example, L1 and L2 pass the == test (they have equivalent values because all their components are equivalent) but fail the is check (they reference two different objects, and hence two different pieces of memory). Notice what happens for short strings, though:

```
>>> S1 = 'text'
>>> S2 = 'text'
>>> S1 == S2, S1 is S2
(True, True)
```

Here, we should again have two distinct objects that happen to have the same value: == should be true, and is should be false. But because Python internally caches and reuses some objects as an optimization, there really is just a single string 'text' in memory, shared by S1 and S2. Hence, the is identity test reports a true result. To trigger the normal behavior, we need to use strings that are longer (or otherwise defeat caching rules prone to change over time):

```
>>> S1 = 'a longer string'
>>> S2 = 'a longer string'
>>> S1 == S2, S1 is S2
(True, False)
```

Of course, because strings are *immutable*, the object caching mechanism is irrelevant to your code—strings can't be changed in place, regardless of how many variables refer to them. If identity tests seem confusing, see [Chapter 6](#) for a refresher on object reference concepts. As a rule of thumb, the == operator is what you will want to use for almost all equality checks; is is reserved for highly specialized roles. You'll see use cases for both later.

As demoed along the way, relative *magnitude* comparisons are also applied recursively to nested data structures:

```
>>> L1 = [1, ('a', 3)]           # Nested 3 > nested 2
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2   # Less, equal, greater: tuple of results
(False, False, True)
```

Here, L1 is greater than L2 because the nested 3 is greater than the nested 2. More broadly, Python compares its core object types as follows:

- *Numbers* are compared by relative magnitude, after conversion to the common highest type if needed (e.g., 1 < 1.1 after 1 is replaced with 1.0).
- *Strings* are compared lexicographically (by the character code-point values returned by ord), and character by character until the end or first mismatch (e.g., 'abc' < 'ac').
- *Lists* and *tuples* are compared by comparing each component from left to right, and recursively for nested structures, until the end or first mismatch (e.g., [1, 3] > [1, 2]).
- *Sets* are equal if both contain the same items (formally, if each is a subset of the other), and magnitude comparison operators for sets apply subset and superset tests.

- *Dictionaries* compare as equal if their sorted (*key*, *value*) lists are equal. Magnitude comparisons are not supported for dictionaries but can be coded by comparing manually sorted `items` results.
- Nonnumeric *mixed-type* magnitude comparisons (e.g., `1 < 'text'`) are errors. By proxy, this also applies to sorts, which use comparisons internally: nonnumeric mixed-type collections cannot be sorted sans conversions.

In general, comparisons of structured objects proceed as though you had written the objects as literals and compared all their parts one at a time from left to right. In later chapters, you'll also see that class-based objects can change the way they are compared. Here, the following sections provide a few more details on Python's built-in comparisons.

Mixed-type comparisons and sorts

The preceding section's last bullet point applies only to nonnumeric mixed-type magnitude tests, not equality, but it also applies by proxy to *sorting*, which does magnitude testing internally. Python disallows mixed-type magnitude testing, except for numeric types and manually converted types:

```
>>> 11 == '11'                                # Equality works but magnitude does not
False
>>> 11 >= '11'
TypeError: '>=' not supported between instances of 'int' and 'str'

>>> ['11', '22'].sort()                        # Ditto for sorts
>>> [11, '11'].sort()
TypeError: '<' not supported between instances of 'str' and 'int'

>>> 11 > 9.123                                  # Mixed numbers convert to highest type
True
>>> str(11) >= '11', 11 >= int('11')          # Manual conversions force the issue
(True, True)

>>> [11, '11'].sort(key=str)                   # Ditto for sorts: see Chapter 8
```

Dictionary comparisons

As noted in [Chapter 8](#), magnitude comparisons don't work for dictionaries directly. Though subject to implementation morph, this purportedly reflects that fact that magnitude comparison would incur too much overhead and may hamper the more common equality test, which may use an optimized scheme that doesn't compare sorted key/value lists:

```
>>> D1 = {'b':3, 'a':1}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2                                    # Equality, not magnitude
True
>>> D1 < D2
TypeError: '<' not supported between instances of 'dict' and 'dict'
```

To work around this limitation, either write loops to compare values by key, or, as also shown in [Chapter 8](#), simply compare sorted key/value lists manually by combining the `items` dictionary method and sorted built-in:


```
>>> list(D1.items())
[('b', 3), ('a', 1)]
>>> sorted(D1.items())
[('a', 1), ('b', 3)]

>>> sorted(D1.items()) < sorted(D2.items())      # Dictionary magnitude tests
False
>>> sorted(D1.items()) >= sorted(D2.items())
True
```

This takes more code than a simple `<` or `>` and may run relatively slowly; but in practice, most programs requiring this behavior either will develop more efficient ways to compare data in dictionaries or won't care about the sloth.

The Meaning of True and False in Python

Notice that the test results returned in the last two examples represent true and false values. They print as the words `True` and `False`, but now that we're using logical tests like these in earnest, it's time to be a bit more formal about what these names really mean.

In Python, as in most programming languages, an integer `0` represents false, and an integer `1` represents true (a heritage rooted in the digital nature of computer hardware). In addition, though, Python recognizes any *empty* data structure as false and any *nonempty* data structure as true. More generally, the notions of true and false are intrinsic properties of every object in Python—each object is either true or false, as follows:

- Numbers are false if zero, and true otherwise.
- Collection objects are false if empty, and true otherwise.
- The `None` placeholder object is always false.
- `True` and `False` are preset to true and false, respectively.

Table 9-4 gives examples of true and false values of various objects in Python.

Table 9-4. Example object truth values

Object	Value
'text'	True
''	False
[1, 2]	True
[]	False
{'a': 1}	True
{}	False
1	True
0.0	False
None	False

As one application of this, because objects are true or false themselves, it's common to see Python programmers code tests like `if X:`, which, assuming `X` is a string, is the same as `if X != ''`. In other words, you can test the object *itself* to see if it contains anything, instead of comparing it to an empty—and therefore false—object of the same type.

The None object

As shown in the last row in [Table 9-4](#), Python also provides a special object called `None`, which is always considered to be false. `None` was introduced briefly in [Chapter 4](#); it is the only value of a special data type in Python and typically serves as an empty placeholder (much like a `NULL` pointer in C).

For example, recall that for lists you cannot assign to an offset unless that offset already exists—the list does not magically grow if you attempt an out-of-bounds assignment. To preallocate a list such that you can store values in any of its offsets, you can fill it with `None` objects:

```
>>> size = 50
>>> L = []
>>> L[size - 1] = 'NO'
IndexError: list assignment index out of range

>>> L = [None] * size
>>> L[size - 1] = 'OK'
>>> L[-10:]
[None, None, None, None, None, None, None, None, None, 'OK']
```

This doesn't limit the size of the list (it can still grow and shrink later), but simply presets an initial size to allow for future index assignments. You could initialize a list with zeros the same way, of course, but best practice suggests using `None` if the type of the list's contents is variable or not yet known.

Keep in mind that `None` does not mean “undefined.” That is, `None` is something, not nothing (despite its name!)—it is a real object and a real piece of memory that is created and given a built-in name by Python itself. Watch for other uses of this special object later in the book; as you'll learn in [Part IV](#), it is also the default return value of functions that don't exit by running into a `return` statement with a result value.

The bool type

While we're on the topic of truth, also keep in mind that the Python Boolean type `bool`, introduced in [Chapter 5](#), simply augments the notions of true and false in Python. As we learned earlier, the built-in words `True` and `False` are just customized versions of the integers 1 and 0—it's as if these two words have been preassigned to 1 and 0 everywhere in Python. Because of the way this new type is implemented, this is really just a minor extension to the notions of true and false already described, designed to make truth values more explicit:

- When used explicitly in truth test code, the words `True` and `False` are equivalent to 1 and 0, respectively, but they make the programmer's intent clearer.
- Results of Boolean tests run interactively print as the words `True` and `False`, instead of as 1 and 0, to make the type of result clearer.

You are not required to use only Boolean types in logical statements such as `if`; all objects are still inherently true or false, and all the Boolean concepts mentioned in this chapter still work as described if you use other types. Python also provides a `bool` built-in function that can be used to extract the Boolean value of an object. You can use this to explicitly check if an object is true—that is, nonzero or nonempty:

```
>>> bool(1)
True
>>> bool('text')
True
>>> bool({})
False
```

In practice, though, you'll rarely notice the Boolean type produced by logic tests, because Boolean results are used automatically by `if` statements and other selection tools. We'll explore Booleans further when we study logical statements in [Chapter 12](#).

Python's Type Hierarchies

As a summary and reference, [Figure 9-3](#) sketches all the major built-in object types available in Python and their relationships. We've explored the most prominent of these in this part of the book. Other objects in [Figure 9-3](#) are program units (e.g., functions and modules) or interpreter internals (e.g., stack frames and compiled code).

The main point to notice here is that *everything* processed in a Python program is an object type. This is sometimes called a “first class” object model, because all objects are on equal footing with respect to your code. For instance, you can pass a class to a function, assign it to a variable, stuff it in a list or dictionary, and so on.

Type Objects

In fact, even types themselves are an object type in Python: the type of an object is an object of type `type` (and not just because it's a decent tongue twister!). Seriously, a call to the built-in function `type(X)` returns the type object of object `X`. The practical application of this is that type objects can be used for manual type comparisons in Python `if` statements. However, for reasons introduced in [Chapter 4](#) that we won't rehash here, manual type testing is usually not the right thing to do in Python, since it limits your code's flexibility. Python is about flexibility, not constraints.

One note on type names: each core type has a built-in name that supports various roles, including type customization through object-oriented subclassing: `dict`, `list`, `str`, `tuple`, `int`, `float`, `complex`, `bytes`, `type`, `set`, and more. Technically speaking, these names reference classes, and calls to these names are really object constructor calls, not simply conversion functions, though you can treat them as simple functions for basic usage.

In addition, the `types` standard-library module provides additional type names for types that are not available as built-ins (e.g., `types.FunctionType` is the type of functions), and the `isinstance` built-in function checks types with consideration of inheritance in OOP—a topic we'll reach later on our Python journey. Because types can be customized with OOP in Python, though, the `isinstance` technique is generally recommended in the very rare cases where code must know about specific types. There's more on type customizations in [Chapter 32](#), and an example in which `isinstance` is useful and warranted in [Chapter 19](#).

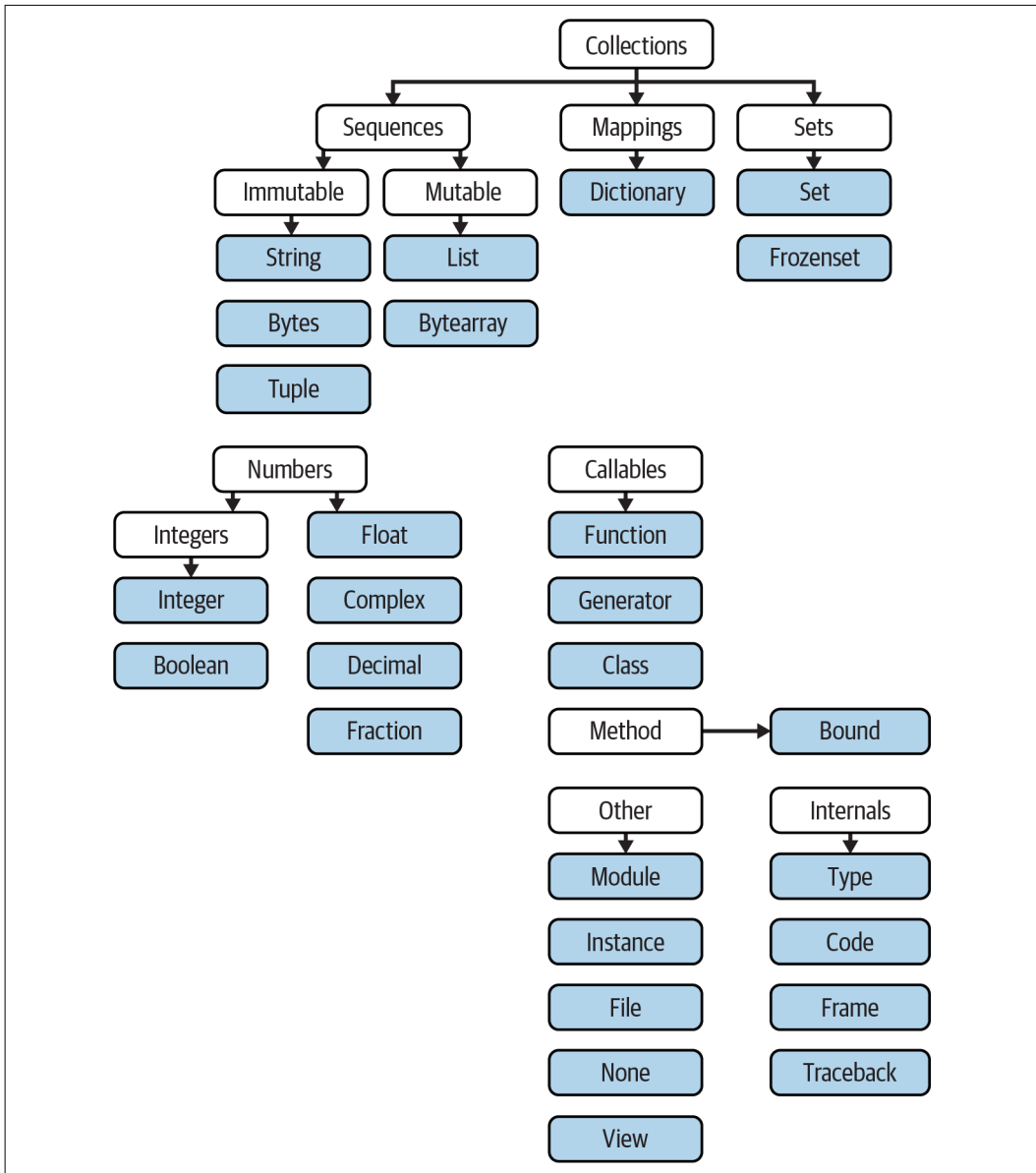


Figure 9-3. Python’s major built-in object types, organized by categories

Other Types in Python

Besides the core objects studied in this part of the book, and the program-unit objects such as functions, modules, and classes that you’ll meet later, a typical Python installation has dozens of additional object types available as linked-in C extensions or imported Python classes—regular expression objects, GUI widgets, network sockets, and so on. Depending on whom you ask, the *named tuple* you met earlier in this chapter may fall in this category too, along with `Decimal` and `Fraction` of [Chapter 5](#).

The main difference between these extra tools and the built-in types you’ve seen so far is that the built-ins have language-defined syntax for creating their objects (e.g., 4 for an integer, [1,2] for a list, the open function for files, and def and lambda for functions). Other tools are made available in standard-library modules that you import to use. For instance, to make a regular-expression object in pattern matching, you import re and call re.compile().

Because most objects in this noncore category are application-level tools that are beyond the scope of this language tutorial, be sure to browse Python’s library reference early and often in your coding career for a comprehensive chronicle of all the supplemental tools available to Python programs.



What about range?: Python’s documentation reclassified the built-in range function as a sequence type, along with lists and tuples, but this is an academic sleight of hand that we won’t adhere to in this book. As you’ll see later, range returns an *iterable* object that produces results on demand, not a physically stored sequence. It supports some—but not all—sequence operations (e.g., indexing works but concatenation does not), but even this is an implementation trick, and hardly enough to constitute a new type on the same level as real sequences. Labeling range an “immutable sequence of integers” conflates tool categories and confuses Python learners.

Built-in Type Gotchas

That’s the end of our look at core data types. We’ll wrap up this part of the book with a discussion of common problems that seem to trap new users (and the occasional expert), along with their solutions. Some of this is a review of ideas we’ve already covered, but these issues are important enough to warrant callouts again here.

Assignment Creates References, Not Copies

Yes, this is redundant, but it’s such a common pitfall that it’s worth underscoring one more time: shared references to *mutable* objects can matter. In the following, for instance, the list object assigned to the name L is referenced both from L and from inside the list assigned to the name M. Changing L in place changes what M references, too:

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']           # Embed a reference to L
>>> M
['X', [1, 2, 3], 'Y']

>>> L[1] = 0                    # Changes M (what M references) too
>>> M
['X', [1, 0, 3], 'Y']
```

This effect usually becomes important only in larger programs, and shared references are often exactly what you want. If objects change out from under you in ways unexpected and unwanted, though, you can avoid sharing objects easily by copying them explicitly. For lists, you can always make a top-level copy by using an empty-limits slice, among other techniques described earlier in this chapter:

```
>>> L = [1, 2, 3]
>>> M = ['X', L[:], 'Y']       # Embed a copy of L
>>> L[1] = 0                   # Changes only L, not M
>>> L
[1, 0, 3]
```

```
>>> M
['X', [1, 2, 3], 'Y']
```

Remember, slice limits default to 0 and the length of the sequence being sliced; if both are omitted, the slice extracts every item in the sequence and so makes a top-level copy (a new, unshared object).

Repetition Adds One Level Deep

As we've learned, repeating a sequence is like adding it to itself a number of times. However, when *mutable* sequences are nested, the effect might not always be what you expect. For instance, in the following example X is assigned to L repeated four times, whereas Y is assigned to a list *containing* L repeated four times:

```
>>> L = [4, 5, 6]
>>> X = L * 4           # Like [4, 5, 6] + [4, 5, 6] + ...
>>> Y = [L] * 4         # [L] + [L] + ... = [L, L, ...]

>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Subtly, because L was nested in the second repetition, Y winds up embedding references back to the *original* list assigned to L, and so is open to the same sorts of side effects noted in the preceding section:

```
>>> L[1] = 0           # Impacts Y but not X
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

This may seem artificial and academic—until it happens unexpectedly in your code! The same solutions to this problem apply here as in the previous section, as this is really just another way to create the shared mutable object reference case—make copies when you don't want shared references:

```
>>> L = [4, 5, 6]
>>> Y = [list(L)] * 4   # Embed a (shared) copy of L
>>> L[1] = 0
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Even more subtly, although Y doesn't share an object with L anymore, it still embeds four references to the *same copy* of it. If you must avoid that sharing too, you'll want to make sure each embedded copy is unique:

```
>>> Y[0][1] = 99       # All four copies are still the same
>>> Y
[[4, 99, 6], [4, 99, 6], [4, 99, 6], [4, 99, 6]]

>>> L = [4, 5, 6]
>>> Y = [list(L) for i in range(4)]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]

>>> Y[0][1] = 99       # And now they're not!
>>> Y
[[4, 99, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

If you remember that repetition, concatenation, and slicing copy only the top level of their operand objects, these sorts of cases make much more sense.

Beware of Cyclic Data Structures

We encountered this concept in a prior exercise but didn't explore it much: if a collection object contains a reference to itself, it's called a *cyclic object*. Python prints a [...] whenever it detects a cycle in the object back to itself, rather than getting stuck in an infinite loop (as it once did long ago, when dinosaurs roamed the planet):

```
>>> L = ['stuff']           # Append reference to same object
>>> L.append(L)             # Makes a cycle back to the same object: [...]
>>> L
['stuff', [...]]
```

Besides understanding that the three dots in square brackets represent a cycle in the object (should they ever crop up in your outputs or job interviews!), this case is worth knowing about because it can lead to real gotchas—cyclic structures may cause code of your own to fall into unexpected loops if you don't anticipate them.

For instance, some programs that walk through structured data must keep a list, dictionary, or set of *already visited* items, and check it when they're about to step into a cycle that could cause an unwanted loop. See the Part I exercise solutions in “Part I, Getting Started” on page 1157 in Appendix B for more on this problem. Also watch for general discussion of recursion in Chapter 19, as well as the *reloadall.py* program in Chapter 25 and the `ListTree` class in Chapter 31 for concrete examples of programs where cycle detection can matter.

As is so often the case in programming, the solution is knowledge: don't use cyclic references unless you really need to, and make sure you anticipate them in programs that must care. There are good reasons to create cycles, but unless you have code that knows how to handle them, objects that reference themselves may be more liability than asset. They need not, however, also be a surprise.

Immutable Types Can't Be Changed in Place

And just once more for completeness: you cannot change an immutable object in place. Instead, you construct a new object with slicing, concatenation, and so on, and assign it back to the original reference, if needed:

```
T = (1, 2, 3)

T[2] = 4           # Error!

T = T[:2] + (4,)   # OK: (1, 2, 4)
```

That might seem like extra coding work (and it is), but the upside is that most of the previous gotchas in this section can't happen when you're using immutable objects like tuples and strings; because they can't be changed in place, they are not generally open to the sorts of side effects that can imperil mutable objects like lists and dictionaries.

Chapter Summary

This chapter explored the last two major core object types—the tuple and the file. We learned that tuples support all the usual sequence operations, have just a few methods, do not allow any in-place changes because they are immutable, and are generalized by the named-tuple extension type. We also learned that files are returned by the built-in `open` function and provide methods for reading and writing content of both the text and binary kind.

Along the way we explored how to translate Python objects to and from strings for storing in files, and we looked at `pickle`, `json`, and other modules for advanced roles (object serialization and binary data). Finally, we wrapped up by reviewing some properties common to all object types (e.g., shared references) and went through a list of common mistakes (“gotchas”) in the object-type domain.

In the next part of this book, we’ll shift gears, turning to the topic of *statement syntax*—the way we code processing steps and logic in our scripts. Along the way, this next part explores all of Python’s basic procedural statements. The next chapter kicks off this topic with an introduction to Python’s general syntax model, which is applicable to all statement types. Before moving on, though, take the chapter quiz, and then work through the end-of-part lab exercises to review type concepts. The next part’s statements largely just create and process objects, so make sure you’ve mastered this domain by working through all the exercises before reading on.

Test Your Knowledge: Quiz

1. How can you determine how large a tuple is? Why is this tool located where it is?
2. Write an expression that changes the first item in a tuple. (4, 5, 6) should become (1, 5, 6) in the process.
3. What is the default for the processing mode argument in a file open call?
4. What module might you use to store Python objects in a file without converting them to strings yourself?
5. How might you go about copying all parts of a nested structure at once?
6. When does Python consider an object to be true?

Test Your Knowledge: Answers

1. The built-in `len` function returns the length (number of contained items) for any container object in Python, including tuples. It is a built-in function instead of a type method because it applies to many different types of objects. In general, built-in functions and expressions may span many object types; methods are specific to a single object type, though some method names may be available on more than one type (`index`, for example, works on lists and tuples).
2. Because they are immutable, you can’t really *change* tuples in place, but you can generate a new tuple with the desired value. Given `T = (4, 5, 6)`, you can change the first item by making a new tuple from its parts by slicing and concatenating: `T = (1,) + T[1:]`. (Recall that single-item tuples require a trailing comma.) You could also convert the tuple to a list, change it in place, and convert it back to a tuple, but this is more expensive and is rarely required in practice—simply use a list if you know that the object will require in-place changes.
3. The default for the processing mode argument in a file open call is `'r'`, for reading text input. For input text files, simply pass in the external file’s name or path (unless you also need to customize things like buffering policies or provide a Unicode text encoding to override your platform’s default—as fleshed out in [Chapter 37](#)).
4. The `pickle` module can be used to store Python objects in a file without explicitly converting them to strings. `json` similarly converts a limited set of Python objects to and from strings per the JSON format. The `struct` module is related, but it assumes the data is to be in packed binary format in the file.

5. Import the `copy` module, and call `copy.deepcopy(X)` if you need to copy all parts of a nested structure `X`. This is also rarely needed in practice; references are usually the desired behavior, and shallow copies (e.g., `aList[:]`, `aDict.copy()`, `set(aSet)`) usually suffice for most copies.
6. An object is considered true if it is either a nonzero number or a nonempty collection object. The built-in words `True` and `False` are essentially predefined to have the same meanings as integer 1 and 0, respectively.

Test Your Knowledge: Part II Exercises

This session asks you to get your feet wet with coding built-in object fundamentals. As before, a few new ideas may pop up along the way, so be sure to flip to the answers in “[Part II, Objects and Operations](#)” on page 1160 in [Appendix B](#) when you’re done (or even when you’re not). If you have limited time, consider starting with exercises 10 and 11 (the most practical of the bunch) and then working from first to last as time allows. This is all fundamental material, though, so try to do as many of these as you can; programming is a hands-on activity, and there is no substitute for practicing what you’ve read to make ideas gel.

1. *The basics*: Experiment interactively with the common type operations found in the various operation tables in this part of the book. To get started, bring up the Python interactive interpreter (a REPL of your choosing), type each of the following expressions, and try to explain what’s happening in each case. Note that the semicolon in some of these is being used as a statement separator, to squeeze multiple statements onto a single line: for example, `X=1;X` assigns and then prints a variable (more on statement syntax in the next part of the book). Also remember that a comma between expressions usually builds a tuple, even if there are no enclosing parentheses: `X,Y,Z` is a three-item tuple, which Python prints back to you in parentheses.

```
2 ** 16
2 / 5, 2 / 5.0

'hack' + 'code'
S = 'Python'
'grok ' + S
S * 5
S[0], S[:0], S[1:]

how = 'fun'
'coding %s is %s!' % (S, how)
'coding {} is {}'.format(S, how)
f'coding {S} is {how}!'

('x',)[0]
('x', 'y')[1]

L = [1, 2, 3] + [4, 5, 6]
L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3] + [4, 5, 6])[2:4]
[L[2], L[3]]
L.reverse(); L
L.sort(); L
L.index(4)

{'a': 1, 'b': 2}['b']
D = {'x': 1, 'y': 2, 'z': 3}
D['w'] = 0
```

```
D['x'] + D['w']
D[(1, 2, 3)] = 4
list(D.keys()), list(D.values()), (1, 2, 3) in D

[[], [''], [], (), {}, None]
```

2. *Indexing and slicing*: At the interactive prompt, define a list named `L` that contains four strings or numbers (e.g., `L=[0, 1, 2, 3]`). Then, experiment with the following boundary cases. You may never see these cases in real programs (especially not in the bizarre ways they may appear here!), but they are intended to make you think about the underlying model, and some may be useful in less artificial forms—slicing out of bounds can help, for example, if a sequence is not as long as you expect:

- a. What happens when you try to index out of bounds (e.g., `L[4]`)?
- b. What about slicing out of bounds (e.g., `L[-1000:100]`)?
- c. Finally, how does Python handle it if you try to extract a sequence in reverse, with the lower bound greater than the higher bound (e.g., `L[3:1]`)? Hint: try assigning to this slice (`L[3:1]='?'`), and see where the value is put. Do you think this may be the same phenomenon you saw when slicing out of bounds?

3. *Indexing, slicing, and del*: Define another list `L` with four items, and assign an empty list to one of its offsets (e.g., `L[2]=[]`). What happens? Then, assign an empty list to a slice (`L[2:3]=[]`). What happens now? Recall that slice assignment deletes the slice and inserts the new value where it used to be.

The `del` statement deletes offsets, keys, attributes, and names. Use it on your list to delete an item (e.g., `del L[0]`). What happens if you delete an entire slice (`del L[1:]`)? What happens when you assign a nonsequence to a slice (`L[1:2]=1`)?

4. *Tuple assignment*: Type the following lines:

```
>>> X = 'code'
>>> Y = 'hack'
>>> X, Y = Y, X
```

What do you think is happening to `X` and `Y` when you type this sequence?

5. *Dictionary keys*: Consider the following code fragments:

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
```

You've learned that dictionaries aren't accessed by offsets, so what's going on here? Does the following shed any light on the subject? (Hint: strings, integers, and tuples share which type category?)

```
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. *Dictionary indexing*: Create a dictionary named `D` with three entries, for keys `'a'`, `'b'`, and `'c'`. What happens if you try to index a nonexistent key (`D['d']`)? What does Python do if you try to assign to a nonexistent key `'d'` (e.g., `D['d']='hack'`)? How does this compare to out-of-bounds assignments and references for lists? Does this sound like the rule for variable names?

7. *Generic operations*: Run interactive tests to answer the following questions:
- What happens when you try to use the + operator on different/mixed types (e.g., string + list, list + tuple)?
 - Does + work when one of the operands is a dictionary?
 - Does the append method work for both lists and strings? How about using the keys method on lists? (Hint: what does append assume about its subject object?)
 - Finally, what type of object do you get back when you slice or concatenate two lists or two strings?
8. *String indexing*: Define a string S of four characters: S = 'hack'. Then type the following expression: S[0][0][0][0][0]. Any clue as to what's happening this time? (Hint: recall that a string is a collection of characters, but Python characters are one-character strings.) Does this indexing expression still work if you apply it to a list such as ['h', 'a', 'c', 'k']? Why?
9. *Immutable types*: Define a string S of four characters again: S = 'hack'. Write an assignment that changes the string to 'heck', using only slicing and concatenation. Could you perform the same operation using just indexing and concatenation? How about index assignment?
10. *Nesting*: Write a data structure that represents your personal information: name (first, middle, last), age, job, address, email address, and phone number. You may build the data structure with any combination of built-in object types you like (lists, tuples, dictionaries, strings, numbers). Then, access the individual components of your data structures by indexing. Do some structures make more sense than others for this object?
11. *Files*: Write a script that creates a new output file called *myfile.txt* and writes the string 'Hello file world!' into it. Then write another script that opens *myfile.txt* and reads and prints its contents. Run your two scripts from the system command line (or other script-launcher tool available to you). Does the new file show up in the directory where you ran your scripts? What if you add a different directory path to the filename passed to open? Note: file write methods do not add new-line characters to your strings; add an explicit \n at the end of the string if you want to fully terminate the line in the file.

Statements and Syntax

Introducing Python Statements

Now that you're familiar with Python's built-in objects, this chapter begins our exploration of its fundamental statement forms. Much like the previous part's approach, we'll begin here with a general introduction to statement syntax and follow up with more details about specific statements in the next few chapters.

In simple terms, *statements* are the code we write to tell Python what our programs should do. If, as suggested in [Chapter 4](#), programs “do things with stuff,” then statements are the way we specify what sort of *things* a program does. Less informally, Python is a procedural, statement-based language; by combining statements, we specify a *procedure* that Python performs to satisfy a program's goals.

The Python Conceptual Hierarchy Revisited

Another way to understand the role of statements is to revisit the concept hierarchy introduced in [Chapter 4](#), which talked about built-in objects and the expressions used to manipulate them. This chapter climbs the hierarchy to the next level of Python program structure:

1. Programs are composed of modules.
2. Modules contain statements.
3. *Statements contain expressions.*
4. Expressions create and process objects.

At their base, programs written in the Python language are composed of statements and expressions. Expressions process objects and are embedded in statements. Statements code the larger *logic* of a program's operation—they use and direct expressions to process the objects we studied in the preceding chapters. Moreover, statements are where objects spring into existence (e.g., in expressions within assignment statements), and some statements create entirely new kinds of objects (functions, classes, and so on). At the top, statements always exist in modules, which themselves are managed with statements.

Python's Statements

Table 10-1 summarizes Python's statement set. Each statement in Python has its own specific purpose and its own specific *syntax*—the rules that define its structure—though, as you'll see, many share common syntax patterns, and some statements' roles overlap. **Table 10-1** also gives examples of each statement, when coded according to its syntax rules. In your programs, these units of code can perform actions, repeat tasks, make choices, build larger program structures, and so on.

This part of the book deals with entries in the table from the top through `break` and `continue`. You've informally been introduced to a few of the statements in **Table 10-1** already; this part of the book will fill in details that were skipped earlier, introduce the rest of Python's procedural statement set, and cover the overall syntax model. Statements lower in **Table 10-1** that have to do with larger program units—functions, classes, modules, and exceptions—lead to larger programming ideas, so they will each have a section of their own. More focused statements (like `del`, which deletes various components) are covered elsewhere in this book, as well as in Python's standard manuals.

Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	<pre>a, b = 'python', 3.12 a = (b := next()) + more</pre>
Calls and other expressions	Running functions	<pre>log.write('app crash')</pre>
<code>print</code>	Printing objects	<pre>print(hack, code, file=log)</pre>
<code>if/elif/else</code>	Selecting actions	<pre>if 'python' in text: read(text)</pre>
<code>match/case</code>	Multiway selections	<pre>match edition: case 6: print(2024)</pre>
<code>for/else</code>	Iteration	<pre>for x in myiterable: print(x)</pre>
<code>while/else</code>	General loops	<pre>while x := file.readline(): print(x)</pre>
<code>pass</code>	Empty placeholder	<pre>if 'python' not in text: pass</pre>
<code>break</code>	Loop exit	<pre>while True: if exittest(): break</pre>
<code>continue</code>	Loop continue	<pre>while True: if skiptest(): continue</pre>
<code>def</code>	Functions and methods	<pre>def f(a, b, c=2, *more): print(a + b + c)</pre>
<code>return</code>	Functions results	<pre>def f(a, b, c=2, *more): return a + b + c</pre>
<code>yield</code>	Generator functions	<pre>def gen(n): for i in n: yield i*2</pre>
<code>global</code>	Namespaces	<pre>x = 1 def function(): global x; x = 2</pre>

Statement	Role	Example
<code>nonlocal</code>	Namespaces	<pre>def outer(): x = 1 def inner(): nonlocal x; x = 2</pre>
<code>async</code>	Coroutine designator	<pre>async def consumer(a, b): await producer()</pre>
<code>await</code>	Coroutine transfer	<pre>await asyncio.sleep(1)</pre>
<code>import</code>	Module access	<pre>import sys</pre>
<code>from</code>	Attribute access	<pre>from sys import stdin as f</pre>
<code>class</code>	Building objects	<pre>class Subclass(Superclass): classAttr = [] def method(self): pass</pre>
<code>try/except/finally</code>	Catching exceptions, termination actions	<pre>try: action() except: print('action error')</pre>
<code>raise</code>	Triggering exceptions	<pre>raise EndSearch(location)</pre>
<code>assert</code>	Debugging checks	<pre>assert X > Y, 'X too small'</pre>
<code>with</code>	Context managers	<pre>with open('data') as file: process(file)</pre>
<code>del</code>	Deleting references	<pre>del data[k] del data[i:j] del obj.attr del variable</pre>
<code>type</code>	Type hinting alias	<pre>type vector = list[float]</pre>

Technically, [Table 10-1](#) is sufficient as a quick preview and reference, but it's not quite complete as is. Here are a few fine points about its content:

- Assignment statements come in a wide variety of syntax flavors, described in [Chapter 11](#): basic, sequence, augmented, and more; and named assignment (`:=`) is used as an expression, not a statement.
- `print` is really a built-in function call, and neither a reserved word nor a statement. Because it will nearly always be run as an expression statement, though, and usually on a line by itself, it's generally thought of as a statement type, and will be treated separately in [Chapter 11](#).
- `yield` and `await` are also expressions instead of statements. Like `print`, they're often used as expression statements and so are included in this table, but scripts may also assign or otherwise use their result, as you'll see in [Chapter 20](#). As expressions, `yield` and `await` are also reserved words, unlike `print`.
- Most of the words used in statements and expressions are *reserved* and cannot be used as variables in your code; this includes `and`, `in`, `if`, `for`, `while`, and others. Newer statements use “soft” reserved words that are reserved only when used in the statements to which they belong; this includes `match`, `case`, and `type` (though not `async` and `await`). We'll formalize the full list of reserved words in [Chapter 11](#).

A Tale of Two ifs

Before we delve into the details of any of the concrete statements in [Table 10-1](#), this book wishes to begin our look at Python statement syntax by showing you what you are *not* going to type in Python code.

Consider the following `if` statement, coded in a C-like language:

```
if (x > y) {  
    x = 1;  
    y = 2;  
}
```

This might be a statement in C, C++, Java, JavaScript, or similar. Now, look at the equivalent statement in the Python language:

```
if x > y:  
    x = 1  
    y = 2
```

The first thing that may pop out at you is that the equivalent Python statement is less cluttered—that is, there are fewer syntactic components. This is by design; as a scripting language, one of Python's goals is to make programmers' lives easier by requiring less typing. And less typing also means less room for mistakes.

More specifically, when you compare the two syntax models, you'll notice that Python adds one new thing to the mix but removes three things that are required in C-like languages.

What Python Adds

The one new syntax component in Python is the colon character (`:`). All Python *compound statements*—statements that have other statements nested inside them—follow the same general pattern of a header line terminated in a colon, followed by a nested block of code usually indented underneath the header line, like this:

```
Header line:  
Nested statement block
```

The colon is required, and omitting it is probably the most common coding mistake among new Python programmers (it's certainly one witnessed thousands of times live in Python training classes). In fact, if you are new to Python, you'll almost certainly forget the colon character very soon, if you haven't already. You'll get an error message if you do, and most Python-friendly editors make this mistake easy to spot:

```
>>> if x  
    if x  
      ^  
SyntaxError: expected ':'
```

Including the colon eventually becomes an unconscious habit—so much so that you may start typing colons in your C-like language code, too (generating reams of entertaining error messages from that language's compiler!).

What Python Removes

Although Python requires the extra colon character, there are three things programmers in C-like languages must include that you don't generally have to code in Python.

Parentheses are optional

The first of these is the set of *parentheses* around the tests at the top of some statements:

```
if (x < y)
```

The parentheses here are required by the syntax of many C-like languages. In Python, though, they are not—we simply omit the parentheses, and the statement works the same way:

```
if x < y
```

Technically speaking, because every expression can be enclosed in parentheses, including them will not hurt in this Python code, and they are not treated as an error if present.

But don't do that. You'll be wearing out your keyboard needlessly, and broadcasting to the world that you're a programmer of a C-like language still learning Python (it happens). The “Python way” is to simply omit the parentheses in these kinds of statements altogether.

End-of-line is end of statement

The second and more significant syntax component you won't find in Python code is the *semicolon*. You don't need to terminate statements with semicolons in Python the way you do in C-like languages:

```
x = 1;
```

In Python, the general rule is that the end of a line automatically terminates the statement that appears on that line. In other words, you can leave off the semicolons, and it works the same way:

```
x = 1
```

There are some ways to work around this rule, as you'll see in a moment (for instance, wrapping code in a bracketed structure allows it to span lines). But, in general, you write one statement per line for the vast majority of Python code, and no semicolon is required.

Here, too, if you are pining for your C programming days (and why would you?) you can continue to use semicolons at the end of each statement—the Python language lets you get away with them if they are present, because the semicolon is also a separator when statements are combined.

But don't do that either (really!). Again, doing so tells the world that you're a programmer of a C-like language who still hasn't quite made the switch to Python coding. The Pythonic style is to leave off the semicolons altogether. Judging from students in classes, this seems a tough habit for some veteran programmers to break. But you'll get there; semicolons are useless noise in this role in Python.

End of indentation is end of block

The third and final syntax component that Python removes...and the one that may seem the most unusual to soon-to-be-ex-programmers of C-like languages, until they've used it for 10 minutes and realize it's actually a feature...is that you do not type *anything* explicit in your code to syntactically mark the beginning and end of a nested block of code. You don't need to include `begin/end`, `then/endif`, or `{/}` around the nested block, as you do in C-like languages:

```
if (x > y) {  
    x = 1;  
    y = 2;  
}
```

Instead, in Python, we consistently indent all the statements in a given single nested block the same distance to the right, and Python uses the statements' physical indentation to determine where the block starts and stops:

```
if x > y:
    x = 1
    y = 2
```

This *indentation* means the blank whitespace all the way to the left of the two nested statements here. Python doesn't care *how* you indent (you may use either spaces or tabs), or *how much* you indent (you may use any number of spaces or tabs). In fact, the indentation of one nested block can be totally different from that of another. The syntax rule is only that for a given single nested block, all of its statements must be indented the same distance to the right. If this is not the case, you will get a syntax error, and your code will not run until you repair its indentation to be consistent:

```
>>> if True:
...     a = 1
...     b = 2
...     b = 2
...     ^
```

IndentationError: unindent does not match any outer indentation level

Why Indentation Syntax?

The indentation rule may seem unusual at first glance to programmers accustomed to C-like languages, but it is an intentional feature of Python: it's one of the main ways that Python almost forces programmers to produce uniform, regular, and readable code. It essentially means that you must line up your code vertically, in *columns*, according to its logical structure. The net effect is to make your code more consistent and readable—unlike much of the code written in C-like languages.

To put that more strongly, aligning your code according to its logical structure is a major part of making it readable, and thus reusable and maintainable, by yourself and others. In fact, even if you never use Python after reading this book, you should get into the habit of aligning your code for readability in *any* block-structured language. Python underscores the issue by making this a part of its syntax, but it's an important thing to do in any programming language, and it has a huge impact on the usefulness of your code.

Your experience may vary, but there's a common phenomenon in large, old C++ programs that have been worked on by many programmers over the years. Almost invariably, each programmer has his or her own style for indenting code. For example, a `while` loop coded in the C++ language may have begun its tenure like this:

```
while (x > 0) {
```

Before we even get into indentation, there are three or four ways that programmers can arrange the `{}` braces in a C-like language, and organizations often suffer political battles and standards docs to address the options (which seems more than a little off-topic for the problem to be solved by programming). Be that as it may, here's the scenario often encountered in C++ code. The first person who worked on the code indented the loop four spaces:

```
while (x > 0) {
    -----;
    -----;
```

That person eventually moved on to other projects (or, sadly, management), only to be replaced by someone who liked to indent further to the right:

```

while (x > 0) {
    -----;
    -----;
        -----;
        -----;

```

That person later moved on to other opportunities (ending their reign of coding terror), and someone else picked up the code who liked to indent less:

```

while (x > 0) {
    -----;
    -----;
        -----;
        -----;
-----;
-----;
}

```

And so on. Eventually, the block is terminated by a closing brace (`}`), which of course makes this “block-structured code” (yes, sarcasm). No: in any block-structured language, Python or otherwise, if nested blocks are not indented consistently, they become very difficult for the reader to interpret, change, or reuse, because the code no longer visually reflects its logical meaning. *Readability matters*, and indentation is a major component of readability.

Here is another example that may have burned you in the past if you’ve done much programming in a C-like language. Consider the following statement in C:

```

if (x)
    if (y)
        statement1;
else
    statement2;

```

Which `if` does the `else` here go with? Surprisingly, the `else` is paired with the nested `if` statement (`if (y)`) in C, even though it looks visually as though it is associated with the outer test (`if (x)`). This is a classic pitfall in the C language, and it can lead to the reader completely misinterpreting the code and changing it incorrectly in ways that might not be uncovered until the Mars rover crashes into a giant rock!

This cannot happen in Python—because indentation is significant, the way the code looks is the way it will work. Consider an equivalent Python statement:

```

if x:
    if y:
        statement1
else:
    statement2

```

In this example, the `if` that the `else` lines up with vertically and visually is the one it is associated with logically and behaviorally (the outer `if x`). In a sense, Python is a WYSIWYG language—what you see is what you get—because the way code looks is the way it runs, regardless of who coded it.

If this still isn’t enough to underscore the benefits of Python’s syntax, here’s another anecdote. Some *companies* that develop systems software in the C language, where consistent indentation is not required, enforce it anyhow. It’s not too uncommon for such groups to run automated tools that analyze the indentation used in code, when it is checked into source control systems at the end of the day. If the tools notice that you’ve indented your code inconsistently, you might just receive an automated email about it the next morning—and so might your manager!

The point is that even when a language doesn't require it, good programmers know that consistent use of indentation has a huge impact on code readability and quality. The fact that Python promotes this to the level of syntax is seen by most as a feature of the language.

Also keep in mind that nearly every programmer-friendly *text editor* has built-in support for Python's syntax model. In the IDLE GUI, for example, lines of code are automatically indented when you are typing a nested block; pressing the Backspace key backs up one level of indentation, and you can customize how far to the right IDLE indents statements in a nested block. There is no requirement on this: *four spaces* is very common, but it's up to you to decide how and how much you wish to indent (unless your company has endured politics and docs to standardize this too). Indent further to the right for further nested blocks, and less to close the prior block.

As a caution, though, you probably shouldn't *mix* tabs and spaces in the same block in Python, unless you do so consistently; use tabs or spaces in a given block, but not both (in fact, Python issues an error for inconsistent use of tabs and spaces, as you'll see in [Chapter 12](#)). Then again, you probably shouldn't mix tabs or spaces in indentation in *any* structured language—such code can cause major readability issues if the next programmer has her or his editor set to display tabs differently than yours. C-like languages might let coders get away with this, but they really shouldn't: the result can be a mangled mess.

Regardless of which language you code in, you should be indenting consistently for readability. In fact, if you weren't taught to do this earlier in your career, your teachers did you a disservice. Most programmers—especially those who must read others' code—consider it an asset that Python elevates this to the level of syntax. Moreover, generating tabs instead of braces is no more difficult in practice for tools that must output Python code, and the page breaks that can obscure code nesting in the print versions of books (including this one!) will not be present in the real world of coding.

In sum, if you do what you should be doing in a C-like language anyhow, but get rid of the braces, your code will satisfy Python's syntax rules.

A Few Special Cases

As mentioned previously, in Python's syntax model:

- The end of a line terminates the statement on that line (without semicolons).
- Nested statements are blocked and associated by their physical indentation (without braces).

Those rules cover almost all Python code you'll write or see in practice. However, Python also provides some special-purpose rules that allow for flexibility in both statements and nested statement blocks. They're not required and should be used sparingly, but programmers have found them useful in practice.

Statement rule special cases

There are three special rules for statements, two of which have already been introduced and used in this book. First of all, although statements normally appear one per line, it is possible to *squeeze* more than one statement onto a single line in Python by separating them with semicolons:

```
a = 1; b = 2; print(a + b)           # Three statements on one line
```

This is the only place in Python where semicolons are required: as statement separators. This only works, though, if the statements thus combined are not *themselves* compound statements. In other words, you can chain together only simple statements, like assignments, and calls to `print` and other functions and methods. Compound statements like `if` tests and `while` loops must still appear on lines

of their own (otherwise, you could squeeze an entire program onto one line, which probably would not make you very popular among your coworkers!).

The other special rule for statements is essentially the inverse: you can make a single statement *span* across multiple lines. To make this work, you simply have to enclose part of your statement in a bracketed pair—parentheses (`()`), square brackets (`[]`), or curly braces (`{}`). Any code enclosed in these constructs can cross multiple lines: your statement doesn’t end until Python reaches the line containing the closing part of the pair. For instance, to continue a list literal:

```
mylist = [1111,          # Continuation lines
          2222,          # Any code in (), [], {}
          3333]
```

Because this code is enclosed in a square brackets pair, Python simply keeps reading on the next line until it encounters the closing bracket. The curly braces surrounding dictionaries (as well as set literals and dictionary and set comprehensions) allow them to span lines this way, too, and parentheses handle tuples, function calls, and expressions. The *indentation* of the continuation lines does not matter, though common sense dictates that the lines should be aligned somehow for readability. Any *# comments* are ignored as usual in continuation lines too, and *nested* brackets must all be closed before the continuation-line run ends.

Parentheses are the catchall device—because *any* expression can be wrapped in them, simply inserting a left *parenthesis* allows you to drop down to the next line and continue your statement:

```
X = (A + B +
     C + D)
```

This technique works within *compound* statements, too, by the way. Anywhere you need to code a large expression, simply wrap it in parentheses to continue it on the next line:

```
if (A == 1 and
    B == 2 and
    C == 3):
    print('hack' * 3)
```

An older rule also allows for continuation lines when the prior line ends in a *backslash*:

```
X = A + B + \
    C + D          # An error-prone older alternative
```

This alternative technique is somewhat discouraged today because it’s difficult to notice and maintain the backslashes. It’s also fairly *brittle* and *error-prone*—there can be no spaces or *# comments* after the backslash, and accidentally omitting it can have unexpected effects if the next line is mistaken to be a new statement (in this example, “*C + D*” is a valid statement by itself if it’s not indented and would silently be run as such). This rule is also a throwback to the C language, where it is commonly used in “*#define*” macros. While `\` may be occasionally useful, when in Pythonland, do as Pythoners do: use bracketed pairs instead of `\` as a rule.

Block rule special case

As mentioned previously, statements in a nested block of code are normally associated by being indented the same amount to the right. As one special case here, and another we met in earlier chapters, the body of a compound statement can instead appear on the same line as the header in Python, after the colon:

```
if x > y: print(x)
```

This allows us to code single-line `if` statements, single-line `while` and `for` loops, and so on. Much like `;` separators, though, this will work only if the body of the compound statement itself does not *contain* any compound statements. That is, only simple statements—assignments, calls to `print` and others, and the like—are allowed after the colon. Larger statements must still appear on lines by themselves. Extra parts of compound statements (such as the `else` part of an `if`, which you’ll meet in the next section) must also be on separate lines of their own. Compound statement bodies can also consist of multiple simple statements separated by semicolons, but this tends to be frowned upon.

In general, even though it’s not always required, if you keep most of your statements on individual lines and indent your nested blocks as a norm, your code will be easier to read and change in the future. Moreover, some code profiling and coverage tools may not be able to distinguish between multiple statements squeezed onto a single line, or the header and body of a one-line compound statement. It is almost always to your advantage to keep things simple in Python. You can use the special-case exceptions to write Python code that’s hard to read, but it takes a lot of work, and there are probably better ways to spend your time.

To see a prime and common exception to one of these rules in action, however (the use of a single-line `if` statement to break out of a loop), and to introduce more of Python’s syntax, let’s move on to the next section and write some real code.

A Quick Example: Interactive Loops

You’ll see all these syntax rules in action when we tour Python’s specific compound statements in the next few chapters, but they work the same everywhere in the Python language. To get started, let’s work through a brief but realistic example that demos the way that statement syntax and nesting come together and introduces a few statements along the way. To work along, either copy and paste this section’s examples from *emedia* into your REPL or run them from the file *interact.py* located in this book’s examples package using any of the launch tools we studied in [Chapter 3](#).

A Simple Interactive Loop

Suppose you’re asked to write a Python program that interacts with a user in a console window. Maybe you’re accepting inputs to send to a database or reading numbers to be used in a calculation. Regardless of the purpose, you need to code a loop that reads one or more inputs from a user typing on a keyboard and prints back a result for each. In other words, you need to write a classic read/evaluate/print loop program, similar to Python’s standard REPL.

In Python, typical boilerplate code for such an interactive loop might look like this:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(reply.upper())
```

This code makes use of a few new ideas and some we’ve already explored:

- The code leverages the Python `while` loop, Python’s most general looping statement. We’ll study the `while` statement in more detail later, but in short, it consists of the word `while`, followed by an expression that is interpreted as a true or false result, followed by a nested block of code that is repeated while the test at the top is true. The word `True` here is considered always true, so this loop continues forever, unless somehow stopped.

- The `input` built-in function we met earlier in the book (to keep windows open after clicks in [Chapter 3](#) and the [Appendix A](#) coverage it referenced) is used here for general console input—it prints its optional argument string as a prompt and returns the user’s typed reply as a string.
- A single-line `if` statement that makes use of the special rule for nested blocks also appears here: the body of the `if` appears on the header line after the colon instead of being indented on a new line underneath it. This would work either way, but as it’s coded, we’ve saved an extra line.
- Finally, the Python `break` statement is used to exit the loop immediately—it simply jumps out of the loop statement altogether, and the program continues after the loop. Without this exit statement, the `while` would loop forever, as its test is always true.

In effect, this combination of statements essentially means “read a line from the user and print it in uppercase until the user enters the word ‘stop.’” There are other ways to code such a loop (e.g., see the note ahead), but the form used here is very common in Python code and serves to illustrate syntax basics.

Notice that all three lines nested under the `while` header line are indented the same amount: because they line up vertically in a column this way, they are the block of code that is associated with the `while` test and repeated. Either a lesser-indented statement or the end of the source file (as here) will suffice to terminate the loop body block.

When this code is run, either interactively or as a script file, here is the sort of interaction we get:

```
Enter text:python
PYTHON
Enter text:312
312
Enter text:stop
```



Or crunch code with the `:=` operator: Spoiler alert—this section’s code is traditional and simple and works as a syntax demo, but it’s possible to reduce it from four lines to two with the `:=` named-assignment expression added in Python 3.8 and covered in the next chapter. This expression assigns a name to another expression’s result, but also returns the assigned value as its overall result. The net effect lets us fetch, assign, and test input on the same line—and all in the loop’s header:

```
while (reply := input('Enter text:')) != 'stop':
    print(reply.upper())
```

While useful in narrow roles, this expression also requires nested parentheses in this context and is arguably more implicit than traditional forms (though ex-C programmers’ mileage may vary!).

Doing Math on User Inputs

Our script works, but now suppose that instead of converting a text string to uppercase, we want to do some math with numeric input—squaring it, for example (perhaps in some misguided effort of an age-input program to tease its users). We might try statements like these to achieve the desired effect:

```
>>> reply = '40'
>>> reply ** 2
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

This won't quite work in our script, though: input from a user is always returned as a *string*, and as discussed in the prior part of this book, Python won't convert object types in expressions unless they are all numeric. We cannot raise a string of digits to a power unless we convert it manually to an integer:

```
>>> int(reply) ** 2
1600
```

Armed with this information, we can now recode our loop to perform the necessary math. Type the following in a file to run it with command line, IDLE menu options, or any other technique we met in [Chapter 3](#) (the REPL both requires a blank line after the `while`, and runs just one statement at a time—the final `print` here wouldn't make sense):

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(int(reply) ** 2)
print('Bye')
```

This script uses a single-line `if` statement to exit on “stop” as before, but it also converts inputs to perform the required math. This version also adds an exit message at the bottom. Because the `print` statement in the last line is not indented *as much* as the nested block of code, it is not considered part of the loop body and will run only once, after the loop is exited:

```
Enter text:2
4
Enter text:40
1600
Enter text:stop
Bye
```

Handling Errors by Testing Inputs

So far so good, but notice what happens when the input is invalid:

```
Enter text:xxx
ValueError: invalid literal for int() with base 10: 'xxx'
```

The built-in `int` function raises an *exception* (i.e., flags an error) here in the face of a nonnumber. If we want our script to be robust, we can check the string's content ahead of time with the string object's `isdigit` method:

```
>>> S = '40'
>>> T = 'xxx'
>>> S.isdigit(), T.isdigit()
(True, False)
```

This also gives us an excuse to further nest the statements in our example. The following new version of our interactive script uses a full-blown `if` statement to work around the exception on conversion errors:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
print('Bye')
```

We'll study the `if` statement in more detail in [Chapter 12](#), but it's a fairly lightweight tool for coding logic in scripts. In its full form, it consists of the word `if` followed by a test and an associated block of code; one or more optional `elif` ("else if") tests and code blocks; and an optional `else` part with a block of code at the bottom to serve as a default. Python runs the block of code associated with the first test that is true, working from top to bottom, or the `else` part if all tests are false.

The `if`, `elif`, and `else` parts in the preceding example are associated as part of the same statement because their opening words all line up vertically (i.e., share the same level of indentation). The `if` statement spans from the word `if` to just before the `print` statement on the last line of the script. In turn, the entire `if` block is part of the `while` loop because all of it is indented under the loop's header line. Statement nesting like this is natural once you start using it.

When we run our new script, its code catches errors before they occur and prints an error message before continuing (which you'll probably want to improve before this code is handed over to the quality-assurance team), but "stop" still gets us out, and valid numbers are still squared:

```
Enter text:5
25
Enter text:xxx
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:10
100
Enter text:stop
Bye
```

Handling Errors with `try` Statements

The preceding solution works, but as you'll see later in the book, the most general way to handle errors in Python is to catch and recover from them completely using the Python `try` statement. We'll explore this statement in depth in [Part VII](#) of this book, but as a preview, using a `try` here can lead to code that some might see as simpler than the prior version:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        num = int(reply)
    except:
        print('Bad!' * 8)
    else:
        print(num ** 2)
print('Bye')
```

This version works exactly like the previous one, but we've replaced the explicit error check with code that assumes the conversion will work and wraps it in an exception handler for cases when it doesn't. In other words, rather than detecting an error, we simply respond if one occurs.

This `try` statement is another compound statement and follows the same pattern as `if` and `while`. It's composed of the word `try`, followed by the main block of code (the action we are trying to run), followed by an `except` part that gives the exception handler code and an `else` part to be run if no exception is raised in the `try` part. Python first runs the `try` part, then runs either the `except` part (if an exception occurs) or the `else` part (if no exception occurs).

In terms of statement nesting, because the words `try`, `except`, and `else` are all indented to the same level, they are all considered part of the same single `try` statement. Notice that the `else` part is associated with the `try` here, not the `if`. As we've seen, `else` can appear in `if` statements in Python, but it can

also appear in try statements and loops—its indentation tells you what statement it is a part of. In this case, the try statement spans from the word try through the code indented under the word else, because the else is indented the same as try. The if statement in this code is a one-liner and ends after the break, so the else cannot apply to it.

Supporting Floating-Point Numbers

Again, we'll come back to the try statement later in this book. For now, be aware that because try can be used to intercept any error, it reduces the amount of error-checking code you have to write, and it's a very general approach to dealing with unusual cases. If we're sure that print won't fail, for instance, this example could be even more concise:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        print(int(reply) ** 2)
    except:
        print('Bad!' * 8)
print('Bye')
```

And if we wanted to support input of floating-point numbers instead of just integers, for example, using try would be much easier than manual error testing—we could simply run a float call and catch its exceptions:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        print(float(reply) ** 2)
    except:
        print('Bad!' * 8)
print('Bye')
```

There is no isfloat method for strings today, so this exception-based approach spares us from having to accommodate all possible floating-point syntax in an up-front error check (a nontrivial task, given the many faces of floats!). When coded this way, we can enter a wider variety of numbers, but errors and exits still work as before:

```
Enter text:50
2500.0
Enter text:40.5
1640.25
Enter text:1.23E-100
1.5129e-200
Enter text:hack
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```



Or run *anything* with `eval` and `exec`: Python’s built-in `eval` call, which we used in Chapters 5 and 9 to convert data in strings and files, would work in place of `float` here, too, and would allow input of arbitrary expressions (“2 ** 100” would be a legal, if curious, input, especially if we’re assuming the program is processing ages!). This is a powerful concept that is open to the same security issues mentioned in the prior chapters. If you can’t trust the source of a code string, use more focused and restrictive conversion tools like `int` and `float`.

Python’s `exec`, used in Chapter 3 to run code read from a file, is similar to `eval` (but assumes the string is a statement instead of an expression and has no result), and its `compile` call precompiles frequently used code strings to byte-code objects for speed. Run a `help` on any of these for more details. We’ll also use `exec` to import modules by name string in Chapter 25—an example of its more dynamic roles.

Nesting Code Three Levels Deep

Let’s look at one last mutation of our code. Nesting can take us even further if we need it to—we could, for example, extend our prior integer-only script to branch to one of a set of alternatives based on the relative magnitude of a valid input:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        num = int(reply)
        if num < 20:
            print('low')
        else:
            print(num ** 2)
print('Bye')
```

This version adds an `if` statement nested in the `else` clause of another `if` statement, which is in turn nested in the `while` loop. When code is conditional or repeated like this, we simply indent it further to the right. The net effect is like that of prior versions, but we’ll now print “low” for numbers less than 20:

```
Enter text:19
low
Enter text:20
400
Enter text:hack
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```

Chapter Summary

That concludes our first look at Python statement syntax. This chapter introduced the general rules for coding statements and blocks of code. As you’ve learned, in Python we normally code one statement per line and indent all the statements in a nested block the same amount (indentation is part of Python’s syntax). However, we also looked at a few exceptions to these rules, including continuation lines and

single-line tests and loops. Finally, we put these ideas to work in an interactive script that demonstrated a handful of statements and showed statement syntax in action.

In the next chapter, we'll start to dig deeper by going over each of Python's basic procedural statements in depth. As you'll see, though, all statements follow the same general rules introduced here.

Test Your Knowledge: Quiz

1. What three things are required in a C-like language but omitted in Python?
2. How is a statement normally terminated in Python?
3. How are the statements in a nested block of code normally associated in Python?
4. How can you make a single statement span multiple lines?
5. How can you code a compound statement on a single line?
6. Is there any valid reason to type a semicolon at the end of a statement in Python?
7. What is a try statement for?
8. What is the most common coding mistake among Python beginners?

Test Your Knowledge: Answers

1. C-like languages require parentheses around the tests in some statements, semicolons at the end of each statement, and braces around a nested block of code. Python requires none of these (but adds a `:`).
2. The end of a line terminates the statement that appears on that line. Alternatively, if more than one statement appears on the same line, they can be separated with semicolons; similarly, if a statement spans many lines, you must terminate it by closing a bracketed syntactic pair.
3. The statements (code lines) in a nested block are all indented the same number of tabs or spaces.
4. You can make a statement span many lines by enclosing part of it in parentheses, square brackets, or curly braces; the statement ends when Python sees a line that contains the closing part of the pair.
5. The body of a compound statement can be moved to the header line after the colon, but only if the body consists of only noncompound statements.
6. Only when you need to squeeze more than one statement onto a single line of code. Even then, this works only if all the statements are noncompound, and it's discouraged because it can lead to code that is difficult to read.
7. The try statement is used to catch and recover from exceptions (errors) in a Python script. It's often an alternative to manually checking for errors in code.
8. Forgetting to type the colon character at the end of the header line in a compound statement is the most common beginner's mistake. If you're new to Python and haven't made it yet, you probably will soon!

Assignments, Expressions, and Prints

Now that we've had a first introduction to Python statement syntax, this chapter begins our in-depth tour of specific Python statements. We'll begin with the basics: assignment statements, expression statements, and print operations. We've already seen all of these in action, but here we'll fill in important details we've skipped so far. Although they're relatively simple, as you'll see, there are optional variations for each of these statement types that will come in handy once you begin writing realistic Python programs.

Assignments

We've been using the Python assignment statement for a while to retain objects in examples. In its basic form, you write the *target* of an assignment on the left of an equals sign, and the *object* to be assigned on the right. The target on the left may be a name or object component, and the object on the right can be an arbitrary expression that creates an object. For the most part, assignments are straightforward, but here are a few key properties to note up front:

- **Assignments create object references.** As discussed in [Chapter 6](#), Python assignments store references to objects in names or data structure components. They always create *references* to objects instead of copying the objects. Because of that, Python variables are more like pointers than data storage areas.
- **Names are created when first assigned.** Python creates a variable name the first time you *assign* it a value (i.e., an object reference), so there's no need to predeclare names ahead of time. Some (but not all) data structure slots are created when assigned, too (e.g., dictionary entries, some object attributes). Once assigned, a name is replaced with the value it references whenever it appears in an expression.
- **Names must be assigned before being referenced.** It's an error to use a name to which you haven't yet assigned a value. Python raises an exception if you try, rather than returning some sort of ambiguous *default* value. This turns out to be crucial in Python because names are not predeclared—if Python provided default values for unassigned names used in your program instead of treating them as errors, it would be much more difficult for you to spot name typos in your code.

- **Some operations perform assignments implicitly.** In this section, we’re concerned with the `=` statement and its `:=` expression relative, but assignment occurs in many contexts in Python. For instance, you’ll see later that module imports, function and class definitions, for loop variables, and function arguments are all implicit assignments. Because assignment works the same everywhere it pops up, all these contexts simply *bind* (i.e., assign) names and object components to object references at runtime.

With those preliminaries in hand, let’s move on to the code.

Assignment Syntax Forms

Although assignment is a general and pervasive concept in Python, in this chapter we are primarily interested in assignment *statements*, plus one limited *expression*. Table 11-1 illustrates the different syntax forms available for coding assignments in Python.

Table 11-1. Assignment statement and expression forms

Operation	Interpretation
<code>target = 'Hack'</code>	Basic assignment
<code>code, hack = 'py', 'PY'</code>	Tuple assignment
<code>[code, hack] = ['py', 'PY']</code>	List assignment
<code>a, b, c, d = 'hack'</code>	Sequence assignment
<code>a, *b = 'hack'</code>	Extended-unpacking assignment
<code>code = hack = 'python'</code>	Multiple-target assignment
<code>code += 1, hack *= 2</code>	Augmented assignments
<code>(python := 3.12) + 0.01</code>	Named assignment expression

The *basic* form atop Table 11-1 is by far the most common: binding a single target (a name or data structure component) to a single object (an expression result). In fact, you could get all your work done with this form alone. The other table entries represent forms that are all optional, but that programmers often find convenient in practice:

Tuple and list assignments

The second and third forms in the table are related. When you code a tuple or list on the left side of the `=`, Python *pairs* objects on the right side with targets on the left by *position* and assigns them from left to right. For example, in the second line of Table 11-1, both sides are tuples (sans parentheses), and the names `code` and `hack` are assigned `'py'` and `'PY'`, respectively. The left of the `=` is special syntax but the right is a real object, which is why this is called “unpacking” assignment—components on the right are unpacked into targets on the left.

Sequence assignment

Tuple and list assignments were later generalized into instances of what we now call *sequence assignment*—any sequence of targets can be assigned to any sequence (really, iterable) of values, and Python assigns the items one at a time by position. We can even mix and match the types of the sequences involved. The fourth line in Table 11-1, for example, pairs a tuple of names with a string of characters: `a` is assigned `'h'`, `b` is assigned `'a'`, and so on. Despite this flexibility, the item on the left of the `=` is still a tuple or list of assignment targets.

Extended-unpacking assignment

An even later assignment form allows more flexibility in how we assign portions of a sequence—or other iterable—to a sequence of targets. The fifth line in [Table 11-1](#), for example, matches `a` with the first character in the string on the right, and the *starred* name `b` with the rest: `a` is assigned `'h'`, and `b` is assigned `['a', 'c', 'k']`. This provides an alternative to assigning the results of slicing operations. Starred collectors like this have also somewhat usurped the term *unpacking*, though this is an artifact more historical than technical.

Multiple-target assignment

The sixth line in [Table 11-1](#) shows the multiple-target form of assignment. In this form, Python assigns a reference to the same object (the object farthest to the right) to all the targets on the left. In the table, the names `code` and `hack` are both assigned references to the same string object, `'python'`. The effect is the same as if we had run `hack = 'python'` followed by `code = hack`, as `hack` evaluates to the original string object (i.e., not a separate copy of that object).

Augmented assignments

The second-to-last line in [Table 11-1](#) is an example of *augmented assignment*—a shorthand that combines an expression and an assignment in a concise way. Saying `code += 1`, for example, has the same effect as `code = code + 1`, but the augmented form requires less typing and is generally quicker to run. In addition, if the target of the assignment is *mutable*, an augmented assignment may run even quicker by choosing an *in-place* update operation instead of an object copy. As you'll see, there is one augmented assignment statement for most binary expression operators in Python (even the unused `@`!).

Named assignment expression

New in Python 3.8, the `:=` operator allows you to code assignment as an *expression*, which returns the value it assigns to a name. This expression can be nested in places where assignment statements don't work syntactically, and in common roles allows you to both assign a name and use its value in the same place in your code.

Basic Assignments

Let's turn to examples at the REPL prompt as usual. We've already used [Table 11-1](#)'s *basic* assignment in this book, so you should be familiar with its basics. In short, it assigns a single *target* to a single value, where the target may be a name, index, slice, or attribute, and the value is any expression:

```
$ python3
>>> L = [1, 2]           # Name target
>>> L[0] = 3             # Index target
>>> L[-1:] = [4, 5]      # Slice target
>>> L
[3, 4, 5]
```

Attribute targets crop up for classes; we haven't studied these yet, but the assignment is straightforward:

```
object.attr = L          # Attribute target (see Part VI)
```

Though *names* are common and dominate examples here, any of these four types of assignment targets can be used in any form of assignment, except where noted ahead (named assignment expressions, for example, allow only names).



Name annotations: Basic assignment statements can also have an *annotation* expression introduced by a colon immediately after the target on the left. This statement form allows only a single target, and is used for *type hinting*, described near the end of [Chapter 6](#). As noted there, because this is optional, convoluted, completely unused by Python, and fundamentally at odds with the language's core idiom, we're skipping it in this book. See Python's docs for more details if you ever stumble in the wild onto the curious case of type declarations in a dynamically typed language.

Sequence Assignments

Next up, here are a few simple and comparable examples of tuple and list assignment (a.k.a. *sequence assignment*) in action, unpacking items into individual variables:

```
>>> first = 1                # Basic assignment
>>> second = 2

>>> A, B = first, second     # Tuple assignment
>>> A, B                     # Similar to A = first; B = second
(1, 2)

>>> [C, D] = [first, second] # List assignment
>>> C, D
(1, 2)
```

Notice that we really are coding two tuples in the third line in this interaction—we've just omitted their enclosing parentheses. Python pairs the *values* in the tuple on the right side of the assignment operator with the *variables* in the tuple on the left side and assigns the values one at a time. The same goes when `=` is surrounded by lists.

Tuple assignment leads to a common coding trick in Python that was introduced in a solution to the exercises at the end of [Part II](#). Because Python creates a temporary tuple that saves the original values of the variables on the right while the statement runs, unpacking assignments are also an easy way to *swap* two variables' values without creating a temporary variable of your own—the tuple on the right remembers the prior values of the variables automatically:

```
>>> first = 1
>>> second = 2
>>> first, second = second, first # Tuples: swaps values
>>> first, second                # Like T = first; first = second; second = T
(2, 1)
```

As already noted, the original tuple and list assignment forms in Python were eventually generalized to accept *any* type of sequence (really, *iterable*) on the right as long as it is of the same length as the sequence on the left. You can assign a tuple of values to a list of variables, a string of characters to a tuple of variables, and so on. In all cases, Python assigns items in the sequence on the right to targets in the sequence on the left by position—from left to right:

```
>>> [a, b, c] = (1, 2, 3)      # Assign tuple of values to list of names
>>> a, c
(1, 3)
>>> (a, b, c) = 'ABC'         # Assign string of characters to tuple names
>>> a, c
('A', 'C')
```

More broadly, while the left side of a sequence assignment is still a sequence (a tuple or list of targets), the right side may be any *iterable* object, not just any sequence. This is a more general category that includes collections both physical (e.g., lists) and virtual (e.g., a file's lines), which was first defined in [Chapter 4](#) and has popped up in passing ever since. We'll firm up this term when we explore iterables in [Chapters 14](#) and [20](#), and apply it to unpack a range iterable in the next section. For now, the “sequence” in assignment is best associated with what's on the left of the =.

Advanced sequence-assignment patterns

Although we can mix and match sequence types around the = symbol, we must generally have the *same number* of items on the right as we have targets on the left, or we'll get an error. As you'll see in the next section, Python allows us to be more general with extended-unpacking * syntax, but the number of items in the assignment target and subject must normally match:

```
>>> string = 'TEXT'
>>> a, b, c, d = string                # Same number on both sides
>>> a, b, c, d
('T', 'E', 'X', 'T')

>>> a, b, c = string                  # Error if not
ValueError: too many values to unpack (expected 3)
```

To be more flexible, we can always *slice*. There are a variety of ways to employ slicing to make this last case work:

```
>>> a, b, c = string[0], string[1], string[2:]    # Index and slice
>>> a, b, c
('T', 'E', 'XT')

>>> a, b, c = list(string[:2]) + [string[2:]]     # Slice and concatenate
>>> a, b, c
('T', 'E', 'XT')

>>> a, b = string[:2]                          # Slice and unpack directly
>>> c = string[2:]
>>> a, b, c
('T', 'E', 'XT')

>>> (a, b), c = string[:2], string[2:]          # Nested sequences
>>> a, b, c
('T', 'E', 'XT')
```

As the last example in this interaction demonstrates, we can even assign *nested* sequences, and Python unpacks their parts according to their shape, as expected. In this case, we are assigning a tuple of two items, where the first item is a nested sequence (a string), exactly as though we had coded it this way:

```
>>> ((a, b), c) = ('TE', 'XT')                # Paired by shape and position
>>> a, b, c
('T', 'E', 'XT')
```

Python pairs the first string on the right ('TE') with the first tuple on the left ((a, b)) and assigns one character at a time, before assigning the entire second string ('XT') to the variable c all at once. In this event, the sequence-nesting shape of the object on the left must match that of the object on the right. Nested sequence assignment like this is somewhat rare to see, but it can be convenient for picking out the parts of data structures with known shapes.

For example, you'll see in [Chapter 13](#) that this technique also works in for loops, because loop items are assigned to the target given in the loop header just as if an = statement had been run:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ...           # Simple tuple assignment

for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: ...   # Nested tuple assignment
```

Such nested sequence assignments can also be achieved with the `match` statement of the next chapter, but we'll hold back the details until then. Sequence-unpacking assignments also give rise to another common coding idiom in Python—assigning an integer series to a set of variables:

```
>>> red, green, blue = range(3)
>>> red, blue
(0, 2)
```

This code initializes the three names on the left of `=` to the integers 0, 1, and 2, respectively. Because each name gets a unique value, it's Python's simplest equivalent to the *enumerated* data types you may have seen in other languages. To make sense of this, you need to recall that the `range` built-in function returns an *iterable*, which generates a list of successive integers (wrap it in a `list` if you wish to display its values at the REPL all at once like this):

```
>>> list(range(3))           # list() required for display
[0, 1, 2]
```

This call was previewed briefly in [Chapter 4](#); because `range` is commonly used in for loops, we'll say more about it in [Chapter 13](#). Another place you may see a tuple assignment at work is for splitting a sequence into its front and the rest, in loops like the following's `while`, introduced in the prior chapter:

```
>>> L = [1, 2, 3, 4]
>>> while L:                     # Repeat until L becomes empty (false)
...     front, L = L[0], L[1:]   # See next section for * alternative
...     print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

The tuple assignment in the loop here could be coded as the following two lines instead, but it's often more convenient to string them together:

```
...     front = L[0]
...     L = L[1:]
```

Notice that this code is using the list as a sort of *stack* data structure, which can often also be achieved with the `append` and `pop` methods of list objects; here, `front = L.pop(0)` would have much the same effect as the tuple assignment statement, but it would be an in-place change. You'll learn more about `while` loops, and other (and often better) ways to step through a sequence with `for` loops, in [Chapter 13](#).

Extended-Unpacking Assignments

The prior section demonstrated how to use manual slicing to make sequence assignments more general. In later Pythons, sequence assignment was further extended to make this easier. In short, a *starred target*, `*X`, can be used on the left of `=` in order to specify a more general matching against the iterable on the right—the starred target is assigned a list, which collects all items in the iterable not assigned to other targets. This is especially handy for common coding patterns such as splitting a sequence into its “front” and “rest,” as in the preceding example.

Extended unpacking in action

Let's jump right into an example to demo how this works. As already shown, sequence assignments normally require exactly as many targets on the left as there are items in the object on the right. We get an error if the lengths disagree, unless we manually slice on the right, as shown in the prior section:

```
>>> seq = [1, 2, 3, 4]
>>> a, b, c, d = seq
>>> a, d
(1, 4)

>>> a, b = seq
ValueError: too many values to unpack (expected 2)
```

We can, however, use a single starred target in this example to match more generally. In the following continuation of our interactive session, `a` matches the first item in the sequence, and `b` matches the rest:

```
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

When a starred target is used (like name `b` here), the number of items on the left need not match the length of the iterable on the right (like sequence `seq` here). In fact, the starred target can appear *anywhere* on the left. For instance, in the next interaction `b` matches the last item in the sequence, and `a` matches everything before the last:

```
>>> *a, b = seq
>>> a
[1, 2, 3]
>>> b
4
```

When the starred target appears in the *middle*, it collects everything between the other targets listed. Thus, in the following interaction, `a` and `c` are assigned the first and last items, and `b` gets everything in between them:

```
>>> a, *b, c = seq
>>> a
1
>>> b
[2, 3]
>>> c
4
```

More generally, wherever the starred target shows up, it will be assigned a list that collects every unassigned item at that position:

```
>>> a, b, *c = seq
>>> a
1
>>> b
2
>>> c
[3, 4]
```

Naturally, like normal sequence assignment, extended-unpacking syntax works for any sequence types (really, again, any *iterable*), not just lists. Here it is unpacking characters in a string and an iterable `range`—which generates values `0...3` on demand:

```
>>> a, *b = 'hack'
>>> a, b
('h', ['a', 'c', 'k'])

>>> a, *b, c = 'hack'
>>> a, b, c
(('h', ['a', 'c'], 'k'))

>>> a, *b, c = range(4)
>>> a, b, c
(0, [1, 2], 3)
```

This is similar in spirit to *slicing*, but not exactly the same—a sequence unpacking assignment always returns a *list* for matched items, whereas slicing returns a sequence of the same type as the object sliced:

```
>>> S = 'hack'

>>> a, b, c = S[0], S[1:-1], S[-1]      # Slices are type specific
>>> a, b, c
('h', 'ac', 'k')

>>> a, *b, c = S                        # But * always returns a list
>>> a, b, c
('h', ['a', 'c'], 'k')
```

Finally, the closing example of the prior section becomes even simpler with this extension, since we don't have to manually slice to get the first and rest of the items (though the “rest” *L* always becomes a list after the first ***):

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, *L = L                    # Get first, rest without slicing
...     print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

Boundary cases

Although extended unpacking is flexible, some boundary (atypical) cases are worth noting. First, the starred target may match just a single item, but is always assigned a list:

```
>>> seq = [1, 2, 3, 4]

>>> a, b, c, *d = seq
>>> print(a, b, c, d)
1 2 3 [4]
```

Second, if there is nothing left to match the starred target, it is assigned an empty list, regardless of where it appears. In the following, names *a*, *b*, *c*, and *d* have matched every item in the sequence, but Python assigns *e* an empty list instead of treating this as an error case:

```
>>> a, b, c, d, *e = seq
>>> print(a, b, c, d, e)
1 2 3 4 []

>>> a, b, *e, c, d = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

Errors can still be triggered, though, if there is more than one starred target, if there are too few values and no star (as before), and if the starred target is not itself coded inside a sequence:

```
>>> a, *b, c, *d = seq
SyntaxError: multiple starred expressions in assignment

>>> a, b = seq
ValueError: too many values to unpack (expected 2)

>>> *a = seq
SyntaxError: starred assignment target must be in a list or tuple

>>> *a, = seq           # A one-item tuple sans parentheses
>>> a                   # Same as a = seq, but makes a new copy!
[1, 2, 3, 4]
```

Technically, the single-appearance rule for starred targets on the left of = really applies only to *each* sequence when there is *nesting* on both sides—though you still can’t have more than one per nested sequence:

```
>>> [(a, *b), (c, *d)] = [(1, 2, 3, 4), (5, 6, 7, 8)]
>>> a, b, c, d
(1, [2, 3, 4], 5, [6, 7, 8])

>>> [(a, *b, *x), (c, *d)] = [(1, 2, 3, 4), (5, 6, 7, 8)]
SyntaxError: multiple starred expressions in assignment
```

And finally, any assignment *target* can be starred—though you may be hard-pressed to find some in real code:

```
>>> a, *L = [1, 2, 3]           # Name target
>>> a, L
(1, [2, 3])
>>> a, *L[0] = [4, 5, 6]       # Index target
>>> a, L
(4, [[5, 6], 3])
>>> a, *L[1:] = [7, 8, 9]      # Slice target
>>> a, L
(7, [[5, 6], 8, 9])

>>> class C: ...code...         # See Part VI
>>> a, *C.attr = 'yikes'        # Attribute target
>>> a, C.attr
('y', ['i', 'k', 'e', 's'])
```

A useful convenience

Keep in mind that extended-unpacking assignment is just a convenience (well, in addition to a mouthful). We can usually achieve the same effects with explicit indexing and slicing, but extended unpacking is simpler to code. The common “first, rest” splitting coding pattern, for example, can be coded either way, but slicing is extra work:

```
>>> seq
[1, 2, 3, 4]

>>> a, *b = seq                # First, rest
>>> a, b
(1, [2, 3, 4])

>>> a, b = seq[0], seq[1:]     # First, rest sans *
```

```
>>> a, b
(1, [2, 3, 4])
```

The also-common “rest, last” splitting pattern can similarly be coded either way, but extended-unpacking syntax again requires noticeably fewer keystrokes (and notably fewer neurons):

```
>>> *a, b = seq                      # Rest, last
>>> a, b
([1, 2, 3], 4)

>>> a, b = seq[:-1], seq[-1]        # Rest, last sans *
>>> a, b
([1, 2, 3], 4)
```

Being both simpler and arguably more natural, extended-unpacking syntax is also common in Python code.

Application to for loops

Because the loop variable in the for loop statement can be any assignment target, extended unpacking works here too. We used the for loop iteration tool briefly in [Chapter 4](#) and will study it formally in [Chapter 13](#), but its tie-in here is straightforward: extended-unpacking assignments may show up after the word for, where a simple variable name is more commonly used:

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]: ...
```

When used in this context, on each iteration Python simply assigns the next tuple of values to the tuple of targets. On the first loop, for example, it’s as if we’d run the following assignment statement:

```
a, *b, c = (1, 2, 3, 4)                # b gets [2, 3]
```

The names a, b, and c can be used within the loop’s code to reference the extracted components. In fact, this is really not a special case at all, but just an instance of general assignment at work. As we saw earlier in this chapter, for example, we can do similar in loops with simple sequence (tuple) assignment:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ...    # a, b, c = (1, 2, 3), ...
```

And we can always emulate extended-unpacking assignment behavior by manually slicing:

```
for all in [(1, 2, 3, 4), (5, 6, 7, 8)]: ...
    a, b, c = all[0], all[1:-1], all[-1]
```

Since we haven’t learned enough to get more detailed about the syntax of for loops, we’ll put this topic on the back burner until its reprise in [Chapter 13](#).

The Many Stars of Python

No, this sidebar is not about personalities (which already permeate software culture more than they should). It’s a reference to all the special-case appearances of * that have cropped up in Python over the years, many of which you’ve already seen. Prior to Python 3.5, the special starred-item syntax forms can appear in:

- *Assignments*, where a single *X starred assignment target of any kind (name, index, slice, attribute), and repeatable in nested parts, collects unmatched items in a new list, as covered in this chapter
- *Function headers*, where single *X and **X starred names collect unmatched positional and keyword arguments in a tuple and dictionary, respectively

- *Function calls*, where single **X* and ***X* starred expressions unpack iterables and mappings into individual positional and keyword arguments, respectively

As of Python 3.5, *function calls* (the latter listed item) support any number of **X* and ***X* unpacking expressions, and any number of **X* and ***X* starred expressions can also appear in *object literals*, where they unpack collections of types implied by the literal in which they appear:

```
[x, *iter]      # List:  unpack items in iterables
(x, *iter, y)   # Tuple: ditto, parentheses or not
{*iter, x}      # Set:   ditto, though values are unordered and unique
{x: y, **map}   # Dict:  unpack keys/values in mappings, rightmost dup wins
```

As of Python 3.10, both **X* and ***X* starred names can also show up in the patterns of the `match` statement covered in the next chapter, where they serve roles similar to that in sequence assignment, though only **X* overlaps between the two, and its assignment in `match` is really a side effect of a true-or-false test.

And as of Python 3.11, an `except*` can appear in `try` statements, where it allows multiple handlers to be run to process multiple exceptions wrapped in an exception *group*—an addition that bifurcates the already-convoluted `try` for narrow roles, which we won't meet until [Chapter 35](#).

All that being said, the avenue of stars has been a meandering walk in Python, and it's not impossible that other parts of the language will rise to stardom in the future. Consult the stars for more info.

Multiple-Target Assignments

Simpler than sequence assignment on first glance, a *multiple-target assignment* simply assigns all the given targets to the same object all the way to the right. The following, for example, assigns the three names (variables) `a`, `b`, and `c` to the string `'code'`:

```
>>> a = b = c = 'code'
>>> a, b, c
('code', 'code', 'code')
```

This form is equivalent to—but easier to code and lighter on line count than—these three assignments:

```
>>> c = 'code'
>>> b = c
>>> a = b
```

Multiple-target assignment and shared references

While this seems simple, keep in mind that there is just one object here, shared by all three variables: they all wind up referencing the *same* object in memory. This behavior is worry-free for *immutable* types—for example, when initializing a set of counters to zero (recall that variables must be assigned before they can be used in Python, so you must initialize counters to zero before you can start adding to them):

```
>>> a = b = 0
>>> b = b + 1
>>> a, b
(0, 1)
```

Here, changing `b` changes only `b` because numbers do not support in-place changes. As long as the object assigned is immutable, it's irrelevant if more than one name references it.

As usual, though, we have to be more cautious when initializing variables to an empty *mutable* object such as a list or dictionary:

```
>>> a = b = []
>>> b.append(42)
>>> a, b
([42], [42])
```

This time, because *a* and *b* reference the *same* object, appending to it in place through *b* will impact what we see through *a* as well. This is really just another example of the shared reference phenomenon we first met in [Chapter 6](#). To avoid the issue, initialize mutable objects in separate statements instead, so that each creates a distinct empty object, by running a distinct literal expression:

```
>>> a = []
>>> b = []           # a and b do not share the same object
>>> b.append(42)
>>> a, b
([], [42])
```

A *tuple* assignment like the following has the same effect with more brevity—by running two list-literal expressions, it creates two distinct objects:

```
>>> a, b = [], []    # a and b do not share the same object
```

Augmented Assignments

In addition to the basic, sequence (original and starred), and multiple assignment forms already covered, Python gained all the assignment statement formats listed in [Table 11-2](#) relatively early in its career. Known as *augmented assignments*, and borrowed from the C language, these formats are mostly just shorthand: they imply the combination of a binary (two-operand) expression and an assignment. For instance, the following two formats are functionally equivalent, though the latter may use in-place options to change *X* directly, as you'll see in a moment:

```
X = X + Y           # Basic form
X += Y              # Augmented form
```

Table 11-2. Augmented assignment statements

<i>X</i> += <i>Y</i>	<i>X</i> -= <i>Y</i>	<i>X</i> *= <i>Y</i>	<i>X</i> /= <i>Y</i>	<i>X</i> @= <i>Y</i>
<i>X</i> /= <i>Y</i>	<i>X</i> %= <i>Y</i>	<i>X</i> **= <i>Y</i>	<i>X</i> >=> <i>Y</i>	
<i>X</i> <<= <i>Y</i>	<i>X</i> &= <i>Y</i>	<i>X</i> = <i>Y</i>	<i>X</i> ^= <i>Y</i>	

Augmented assignment works on any type that supports the implied binary expression. For example, here are two ways to add 1 to a name; really, they both change a name to reference different values:

```
>>> x = 1
>>> x = x + 1      # Basic
>>> x
2
>>> x += 1         # Augmented
>>> x
3
```

When applied to a *sequence* such as a string, the augmented form performs concatenation instead, because that's what *+* means for such objects. Thus, the second line here is equivalent to typing the longer *S = S + 'HACK'*:

```
>>> S = 'hack'
>>> S += 'HACK'           # Implied concatenation
>>> S
'hackHACK'
```

As shown in [Table 11-2](#), there are analogous augmented assignment forms for other Python binary expression operators (i.e., operators with values on their left and right sides). For instance, `X *= Y` multiplies and assigns, `X >>= Y` shifts right and assigns, and so on (though per [Chapter 5](#), the form `X @= Y` is unused and unimplemented by Python itself today). All told, augmented assignments have three noteworthy advantages:¹

- There's less for you to type. (Need this book say more?)
- The left side has to be evaluated only once. In `X += Y`, `X` may be a complicated object expression. In the augmented form, its code must be run only once. However, in the long form, `X = X + Y`, `X` appears twice and must be run twice. Because of this, augmented assignments usually run faster.
- The optimal technique is automatically chosen. That is, for objects that support *in-place* changes, the augmented forms automatically perform in-place change operations instead of slower copies.

The last point here requires a bit more explanation. For augmented assignments, in-place operations may be applied for mutable objects as an optimization. Recall that lists can be extended in a variety of ways. To add a single item to the end of a list, we can concatenate or call `append`:

```
>>> L = [1, 2]
>>> L = L + [3]           # Concatenate one: slower
>>> L
[1, 2, 3]
>>> L.append(4)           # Faster, but in place
>>> L
[1, 2, 3, 4]
```

And to add a set of items to the end, we can either concatenate again or call the list `extend` method:

```
>>> L = L + [5, 6]        # Concatenate many: slower
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8])      # Faster, but in place
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

In both cases, concatenation is less prone to the side effects of shared object references but will generally run slower than the in-place equivalent. Concatenation operations must create a new object, copy in the list on the left, and then copy in the list on the right. By contrast, in-place method calls simply add items at the end of a memory block (it can be a bit more complicated than that internally, but this description suffices).

When we use augmented assignment to extend a list, we can largely forget these details—Python automatically calls the quicker `extend` method (or its equivalent) instead of using the slower concatenation operation implied by `+`:

1 C/C++ programmers also take note: although Python now supports statements like `X += Y`, it still does not have C's auto-increment/decrement operators (e.g., `X++`, `--X`). These don't quite map to the Python object model because Python has no notion of *in-place* changes to immutable objects like numbers. As a preview of the next section, there also are no augmented-named expression forms today; `:= 1` would be close to `++` but thankfully is fiction. That said, we eventually got `+=` and `:=`, so...

```
>>> L += [9, 10]           # Mapped to L.extend([9, 10])
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

As suggested in [Chapter 6](#), we can also use slice assignment (e.g., `L[len(L):] = [11,12,13]`), but this works roughly the same as the simpler and more mnemonic list extend method or the `+=` statement. Note, however, that because of this equivalence `+=` for a list is not exactly the same as a `+` and `=` in all cases—for lists `+=` allows arbitrary sequences (just like `extend`), but concatenation normally does not:

```
>>> L = []
>>> L += 'hack'           # += and extend allow any sequence, but + does not!
>>> L
['h', 'a', 'c', 'k']
>>> L = L + 'code'
TypeError: can only concatenate list (not "str") to list
```

Moreover, using *index* and *slice* targets in augmented assignment may change mutables in other ways:

```
>>> L = [1, 2]
>>> L[0] += 10           # Change an item of a list, in place
>>> L
[11, 2]

>>> L[-1:] += [3, 4]     # Change a section of a list, in place
>>> L
[11, 2, 3, 4]
```

Augmented assignment and shared references

More subtly, the *in-place* change implicit in `+=` for lists is very different from the *new* object created by `+` and concatenation. As for all shared-reference cases, this difference may matter when objects are shared by many names:

```
>>> L = [1, 2]
>>> M = L                # L and M reference the same object
>>> L = L + [3, 4]        # Concatenation makes a new object
>>> L, M                  # Changes L but not M
([1, 2, 3, 4], [1, 2])

>>> L = [1, 2]
>>> M = L
>>> L += [3, 4]           # But += really means extend, not +
>>> L, M                  # M sees the in-place change too!
([1, 2, 3, 4], [1, 2, 3, 4])
```

This only matters for mutables like lists and dictionaries, and it is a fairly obscure case (at least, until it impacts your code!). As always, make copies of your mutable objects if you need to break the shared reference structure.

Named Assignment Expressions

For most of Python's three-decade tenure, it resisted emulating the assignment-as-expression idiom of the C language, on the grounds that it was too subtle and error-prone, and fostered code that was hard to read. While these concerns still apply, Python 3.8 gained a flavor of this, known as *named assignment*.

Importantly, this flavor does *not* make normal `=` assignment statements nestable expressions, as they are in C. Instead, it adds a new expression operator to Python, `:=`, on the grounds that its different and

limited syntax may neutralize pitfalls of other languages. You still can't accidentally type `=` when you mean `==`, and `:=` is visually distinct.

We explored this expression briefly in a spoiler note in the previous chapter. In short, the expression `name := value` first evaluates expression `value`, and then both:

- *Assigns* the result to the provided variable `name`
- *Returns* the result as the value of the overall `:=` expression

There's no reason to use this syntax for the first part alone, because all of Python's `=` assignment statements already assign values to names (in fact, you'll get a syntax error if you try using `:=` as a statement sans parentheses). Because `:=` is an *expression* that returns the value assigned, though, it can be nested in contexts where statements aren't allowed and may be used in roles that both test and then use the assigned name.

As an artificial first example, the following nested `:=` both assigns 2 to `b`, and returns it to be used in the `*` string-repetition result assigned to `a`:

```
>>> a = 'hack' * (b := 2)
>>> a
'hackhack'
>>> b
2
```

Among its deliberate limitations, named assignment allows only a simple, single *name* (variable) on the left, so neither other assignment-statement forms nor references to data-structure components work here:

```
>>> (python := 3.12) + 0.01      # Just a name on the left
3.13
>>> python
3.12

>>> (python, Python := 3.12, 3.13) + 0.01
TypeError: can only concatenate tuple (not "float") to tuple
>>> (python[0] := 3.12) + 0.01
SyntaxError: cannot use assignment expressions with subscript
>>> (python.attr := 3.12) + 0.01
SyntaxError: cannot use assignment expressions with attribute
```

The first of these failers, for example, is taken to be a three-item tuple with `:=` in the middle, not a two-target named sequence assignment. Moreover, named assignment does not support any of the *augmented* assignment forms we met earlier: there is no `+=`, for instance, and coding statement `X += 1` may actually take less work than expression `(X := X + 1)` (subtly, `a :=+` runs, but simply applies the `+` identity operator to the expression on the right).

When to use named assignment

While more useful contexts for `:=` are limited, its most common use cases arise in concert with the `if` and `while` statements, which were both introduced in the preceding chapter and earlier. Without `:=`, it was—and still is—common to assign a variable before the statement, test it in the statement header, and then use it in the statement body.

For example, code that reads lines from files and must detect the end of the file (which, you'll recall from [Chapter 9](#), means an empty string that is logically false) may look like these partial snippets (to run code in this section live, open a text file for input and assign it to `file` before each snippet):

```
line = file.readline()          # Sans the := expression
if line:
    print(line)

line = file.readline()          # Ditto, in while loops
while line:
    print(line)
    line = file.readline()
```

To avoid redundant calls, it's also common to code the latter like this in Python—as in the preceding chapter:

```
while True:                      # Sans both := and redundancy
    line = file.readline()
    if not line: break
    print(line)
```

These forms still work well, and the `:=` is never required. With `:=`, however, we can sometimes collapse a fetch, assignment, and test into a one-liner within statement headers themselves:

```
if line := file.readline():      # The := alternatives
    print(line)

while line := file.readline():    # Read all lines (vs: for line in file)
    print(line)
```

In both cases, because `:=` *returns* the results of the `readline` call, its logical value can be tested in the statement header itself. And because `:=` also *assigns* the result to `line`, it can be used in the statement's body if it is run. The upshot is that `:=` provides a sort of *shortcut* that allows multiple operations to be coded in a compact way.

If you opt to use this expression, bear in mind that it often requires enclosing *parentheses* to avoid interacting with surrounding code. Even comparing its result to an explicit value, for instance, mandates parentheses that are not required in the equivalent statements:

```
if (line := file.readline()) != ignore:    # Parentheses required
    print(line)

while (line := file.readline()) != stop:    # And not a bad idea
    print(line)
```

As a rule of thumb, if you opt to use `:=`, wrapping it in parentheses both sets it off visually and avoids sticky issues that may arise if the code surrounding it changes its meaning unexpectedly. In the preceding, for example, names are assigned the results of the `!=` tests if parentheses are omitted, because it binds tighter than `:=` (see [Chapter 9](#)). Moreover, `:=` has special syntax rules that we'll omit here, but they encourage parenthesized usage by design.

Besides file reads and similar roles, the `:=` can be leveraged to reuse results in literals and get last results in comprehensions and other iteration tools, and can even be nested in f-strings and itself—with requisite parentheses:

```
>>> [val := 'Py!', val * 2, val * 3]          # Reusing a value
['Py!', 'Py!Py!', 'Py!Py!Py!']

>>> list(pow := 2 ** num for num in [2, 4, 8]) # Capturing a result
[4, 16, 256]
```

```
>>> pow
256

>>> f'Hello {(name := input('Who are you? '))}'    # Nesting in f-strings (hmm)
Who are you? Pat
'Hello Pat'
>>> name
'Pat'

>>> (x := (y := (z := 1) + 1) + 1)                # Nesting in itself (hmm * 2)
3
>>> x, y, z
(3, 2, 1)
```

As another rule of thumb, bear in mind that a deeply nested and obscure `:=` expression might seem clever but will be a lot harder to read—and even *notice*—than a simple standalone assignment. You’ll be able to judge this for yourself in examples in [Chapter 20](#). Generally speaking, though, for the sake of others (including your future self!), use `:=`, like so many nestable tools, sparingly and wisely. Clarity is usually worth the extra line or two.

Variable Name Rules

Now that we’ve explored assignment statements and expressions, it’s time to get more formal about the use of variable *names*. In Python, names come into existence when you assign values to them, but there are a few rules to follow when choosing names for the subjects of your programs:

Syntax: (underscore or letter) + (any number of letters, digits, or underscores)

Variable names must start with an underscore or letter, which can be followed by any number of letters, digits, or underscores. `_hack`, `hack`, and `Hack_1` are legal names, but `1_hack`, `hack$`, and `@#!` are not.

Python also allows non-ASCII *Unicode* characters to appear in variable names, but such characters may make your code difficult to use in some contexts and are subject to a handful of rules that require deep Unicode knowledge and are too arcane to cover here. See [Chapter 37](#)’s note “Unicode in variable names” for expanded coverage of this topic, and Python’s language manuals for full details.

Case matters: HACK is not the same as hack

Python always pays attention to case in programs, both in names you create and in reserved words. For instance, the names `X` and `x` refer to two different variables. For portability, case also matters in the names of imported module files, even on platforms where the filesystems are case-insensitive (as is common in Windows). That way, your imports still work after programs are copied to differing platforms.

Reserved words are off-limits

Names you define cannot be the same as words that mean special things in the Python language. For instance, Python will raise a syntax error if you try to use `class` as a variable name, but `klass` and `Class` work fine. [Table 11-3](#) lists the words that are currently *reserved*—and hence off-limits for names of your own—in Python. Note: Python’s docs call these “keywords” today, despite the longstanding and differing use of “keywords” for pass-by-name function arguments; to avoid confusion, this book uses “reserved words” for, well, reserved words.

Table 11-3. Python reserved words

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

In addition to [Table 11-3](#) (and as partly leaked in [Chapter 10](#)), the words `match`, `case`, `_`, and `type` are *soft* reserved words: they are reserved only in the context of the statement to which they belong and can be used as variables anywhere else. As you’ll learn in the next chapter, the first three are part of a multiple-choice `match` statement; the latter is used by a `type` statement that creates type aliases, used for the type hinting discussed at the end of [Chapter 6](#).

As you can see, most of Python’s reserved words are all lowercase. They are also all truly reserved—unlike names in the *built-in scope* that you will meet in the next part of this book, you cannot redefine reserved words by assignment. The statement `and = 1`, for instance, results in a syntax error.

Besides being of mixed case, the first three entries in [Table 11-3](#)—`False`, `None`, and `True`—are somewhat unusual in meaning: they also appear in the built-in scope of Python described in [Chapter 17](#), and they are technically names assigned to objects. They are truly reserved in all other senses, though, and cannot be used for any other purpose in your script other than that of the objects they represent. All the other reserved words are hardwired into Python’s syntax and can appear only in the specific contexts for which they are intended.²

Furthermore, because module names in `import` statements become variables in your scripts, variable name constraints extend to your *module filenames* too. For instance, you can code files called *and.py* and *my-code.py* and can run them as top-level scripts, but you cannot import them: their names without their *.py* extensions become *variables* in your code on imports, and so must follow all the variable rules just outlined. Hence, reserved words are off-limits, and dashes won’t work, though underscores will. This module idea will be revisited in [Part V](#) of this book, where you’ll find that this constraint also applies to names of “package” folders by extension.

Python’s Deprecation Policies

Because making new words reserved has the potential to break code widely, such changes are generally phased into the language gradually. When a change might impact existing code, Python often makes it an option and begins issuing *deprecation warnings* one or more releases before the mod is officially enabled. The idea is that you should have time to notice the warnings and update your code before upgrading your Python. More recently, the “soft” reserved-word category has arisen to allow new reserved words to be added with neither warnings nor breakages. This seems a tacit recognition that deprecations only go so far.

² Exception: Alternative implementations of Python such as Jython might allow reserved words to appear as identifiers in some contexts. See [Chapter 2](#) for more on alternative Pythons.

Deprecation policies are not always followed, especially for major releases like 3.0 (which broke existing code wantonly), but also for newer changes deemed to be justifiable or innocuous by their promoters. In theory, changes are overseen by a *steering committee* charged with enforcing deprecation policies. While this helps in some cases, it is largely ineffectual in stemming the flood of opinionated morph and bloat in Python, which makes engineering reliable and durable software more challenging than it should be.

Even when deprecation warnings *are* issued, though, programs that cannot be easily changed will break if they have been distributed in source code form to users who innocently upgrade their local Python. Those on the receiving end of such flux may legitimately see deprecation warnings less as a courtesy than a bandage on a gaping wound. Warning that you're going to be rude doesn't make it OK to be rude!

Naming conventions

Besides these rules, there is also a set of naming *conventions*—rules that are not required but are followed in normal practice. For instance, because names with two leading and trailing underscores (e.g., `__name__`) generally have special meaning to the Python interpreter, you should avoid this pattern for your own names except in contexts where it is expected. Here is a list of the conventions Python follows:

- Names that begin with a single underscore (`_X`) are not imported by a `from module import *` statement, described in [Chapter 23](#).
- Names that have two leading and trailing underscores (`__X__`) are system-defined names that have special meaning to the interpreter and provide implementation details in the user-defined OOP classes of [Part VI](#).
- Names that begin with two underscores and do not *end* with two more (`__X`) are localized (“mangled”) to enclosing classes, per the discussion of pseudoprivate attributes in [Chapter 31](#).
- The name that is just a single underscore (`_`) retains the result of the last expression when you are working interactively at some REPLs and is a soft keyword for a wildcard in `match`, as covered in [Chapter 12](#).

In addition to these Python interpreter conventions, there are various other conventions that Python programmers usually follow. For instance, some programmers distinguish parts of long names using “camelCase” (`aLongName`), and others use underscores (`a_Long_name`); either is completely valid according to both Python and this book.

Later in the book, you'll also see that *class* names commonly start with an uppercase letter and *module* names with a lowercase letter, and that the name `self`, though not reserved, usually has a special role in classes. Moreover, in [Chapter 17](#) we'll study another, larger category of names known as the *built-ins*, which are predefined but not reserved (and so can be reassigned: `open = 99` silently works, though you might occasionally wish it didn't!).

Names have no type, but objects do

This is mostly review, but remember that it's crucial to keep Python's distinction between names and objects clear. As described in [Chapter 6](#), objects have a type (e.g., integer, list) and may be mutable or not. Names (a.k.a. variables), on the other hand, are always just references to objects; they have no

notion of mutability and have no associated type information, apart from the type of the object they happen to reference at a given point in time.

Thus, it's OK to assign the same name to different kinds of objects at different times:

```
>>> x = 0                # x bound to an integer object
>>> x = 'Hello'          # Now it's a string
>>> x = [1, 2, 3]        # And now it's a list
```

Especially when we step up to functions and classes later in this book, you'll see that this generic nature of names can be a decided advantage in Python programming. In [Chapter 17](#), you'll also learn that names live in something called a *scope*, which defines where they can be used; the place where you assign a name determines where it is visible.³

Expression Statements

In Python, you can use any *expression* as a statement, too—which usually means on a line by itself. Because the result of the expression won't be saved, though, it usually makes sense to do so in scripts only if the expression does something useful as a side effect. Expressions are commonly used as statements in two situations:

For calls to functions and methods

Some functions do their work without returning a value. Such functions are sometimes called *procedures* in other languages. Because they don't return values that you might be interested in retaining, you can call these functions with expression statements. This also applies to methods, which are just functions with an implied subject.

For printing values at the interactive prompt

As you certainly know by now, Python echoes back the results of expressions typed at the interactive command line. Technically, these are expression statements, too; they serve as a shorthand for typing `print` statements.

[Table 11-4](#) lists some common expression statement forms in Python. Calls to functions and methods are coded with zero or more argument objects (really, expressions that evaluate to objects) in parentheses, after the function or method name.

Table 11-4. Common Python expression statements

Operation	Interpretation
<code>hack('Py', 3.12)</code>	Function calls
<code>code.hack('Py')</code>	Method calls
<code>hack</code>	Printing results in the interactive interpreter (REPL)
<code>print(a, b, c, sep='')</code>	Printing operations (a special function call)
<code>yield x ** 2</code>	Yielding expression statements (generators)
<code>await producer()</code>	Pausing for steps to finish (coroutines)

³ If you've used a more restrictive language like C++, you may be interested to know that there is no notion of C++'s `const` declaration in Python; certain objects may be *immutable*, but names can always be assigned. Python also has ways to hide names in classes and modules, but they're not the same as C++'s declarations (if hiding attributes matters to you, see the coverage of `_X` module names in [Chapter 25](#), `__X` class names in [Chapter 31](#), and the `Private` and `Public` class decorators example in [Chapter 39](#)).

The last three entries in [Table 11-4](#) are somewhat special cases. As you’ll see later in this chapter, printing in Python is a function call usually coded as a statement by itself, and important enough to callout here. The `yield` and `await` operations on generator and coroutine functions (discussed in [Chapter 20](#)) are regularly coded as statements as well. All three, though, are really just instances of expressions masquerading as statements.

For instance, though you normally run a `print` call on a line by itself as an expression statement, it actually returns a value like any other function call—the return value is `None`, the default return value for functions that don’t return anything meaningful (it requires another `print` to reveal in output):

```
>>> x = print('code')           # print is a function-call expression
code
>>> print(x)                    # But is usually coded as an expression statement
None
```

Also keep in mind that although expressions can appear as statements in Python, the *converse* is not true: statements cannot be used as expressions. A statement that is not also an expression must generally appear on a line all by itself, not nested in a larger syntactic structure.

For example, Python doesn’t allow you to embed basic assignment statements (`=`) in other expressions. The rationale for this is that it avoids common coding mistakes; you can’t accidentally change a variable by typing `=` when you really mean to use the `==` equality test. If you really miss this coding pattern from other languages, though, the newer and visually distinct `:=` named-assignment expression we met earlier works the same way with less chance of being confused with `==`, and a variety of coding alternatives achieve similar goals (e.g., see [while](#) coverage in [Chapter 13](#)).

Expression Statements and In-Place Changes

This brings up a common mistake in Python work, which we’ve encountered before, but is so pervasive that it merits another quick nag here. Expression statements are often used to run list methods that change a list in place:

```
>>> L = [1, 2]
>>> L.append(3)                 # Append is an in-place change
>>> L
[1, 2, 3]
```

It’s not unusual, though, for Python newcomers to code this as an assignment statement instead:

```
>>> L = L.append(4)             # But append returns None, not L
>>> print(L)                   # So we lose our list!
None
```

But this doesn’t work: in-place change methods like `append`, `sort`, and `reverse` always change the list in place, but do not return the list they have changed; instead, they return the `None` object. Assigning such an operation’s result back to the variable name loses your reference to the list (and it’s probably garbage-collected in the process).

So don’t do that—call in-place change operations without assigning their results. We’ll revisit this phenomenon in [“Common Coding Gotchas” on page 372](#), because it can also appear in the context of some looping statements we’ll explore in the chapters ahead.

Print Operations

In Python, `print` prints things—it's simply a programmer-friendly interface to the standard output stream. It's a function-call expression that we're calling out here because it's so pervasive and is usually coded as a statement.

Technically, printing converts one or more objects to their textual representations, adds some minor formatting, and sends the resulting text to either standard output or another file-like stream. In a bit more detail, `print` is strongly bound up with the notions of *files and streams* in Python:

File object methods

In [Chapter 9](#), we explored file object methods that write text (e.g., `file.write(str)`). Printing operations are similar, but more focused—whereas file write methods write strings to arbitrary files, `print` writes objects to the `stdout` stream by default, with some automatic conversion and formatting added. Unlike with file methods, there is no need to convert objects to strings when using print operations.

Standard output stream

The standard output stream (often known as `stdout`) is simply a default place to send a program's text output. Along with the standard input and error streams, it's one of three data connections created when your script starts. The standard output stream is usually mapped to the window where you started your Python program or REPL, unless it's been redirected to a file or pipe in your operating system's shell or rerouted by your coding GUI.

Because the standard output stream is available in Python as the `stdout` file object in the built-in `sys` module (i.e., `sys.stdout`), it's possible to emulate `print` with file write method calls. However, `print` is noticeably simpler to use in most roles and makes it easy to print text to other files and streams.



Blast from the past: Printing is also one of the most visible places where Python 3.X and 2.X diverged: it morphed from statement in 2.X to built-in function in 3.X—and broke nearly every existing Python program in the process. This book no longer covers 2.X's statement, but the legacy of 3.X's backward incompatibilities lives on as a natural brake on language changes (and in the muscle memory of Python coders who still type `print` x unconsciously!).

The print Function

Strictly speaking, printing is not a separate statement form. Instead, it is simply an instance of the *expression statement* we studied in the preceding section. The `print` built-in function is normally called on a line of its own, because it doesn't return any value we care about (technically, it returns `None`, per the preceding section). Because it is a normal function, though, it can use standard function-call syntax (including keyword arguments for special operation modes) and may be both passed around as an object and reassigned to a different implementation.

Call format

Syntactically, calls to the `print` function have the following form:

```
print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout][, flush=False])
```

In this notation, items in square brackets are optional and may be omitted in a given call, and values after `=` give keyword-argument defaults. In English, this built-in function prints the textual representation of one or more objects, separated by the string `sep` and followed by the string `end`, to the stream `file`, flushing buffered output or not per `flush`.

The `sep`, `end`, `file`, and `flush` parts, if present, must be given as *keyword arguments*—that is, you must use a *name=value* syntax to pass the arguments by name instead of position. Keyword arguments are covered in depth in [Chapter 18](#), but they’re straightforward to use. The keyword arguments sent to this call may appear in any left-to-right order following the objects to be printed, and they control the `print` operation:

- `sep` is a string inserted between each object’s text, which defaults to a single space if not passed. Passing an empty string suppresses separators altogether.
- `end` is a string added at the end of the printed text, which defaults to a `\n` newline character if not passed. Passing an empty string avoids dropping down to the next output line at the end of the printed text—the next `print` will keep adding to the end of the current output line.
- `file` specifies the file, standard stream, or other file-like object to which the text will be sent. It defaults to the `sys.stdout` standard output stream if not passed, but any object with a `file-like write(string)` method may be passed, including real file objects that have already been opened for output to external files.
- `flush` defaults to `False`. It allows prints to mandate that their text be flushed through the output stream immediately to any waiting recipients. Normally, whether printed output is buffered in memory or not is determined by the `file` object; passing a true value to `flush` forcibly flushes the stream.

The textual representation of each *object* to be printed is obtained by passing the object to the `str` built-in call (or its equivalent inside Python); as we’ve seen, this built-in returns a “user friendly” display string for any object.⁴ With no arguments at all, the `print` function simply prints a newline character to the standard output stream, which usually displays a blank line.

The print function in action

Printing is probably simpler than its full details may imply. To illustrate, let’s run some quick examples in the REPL. The following prints a variety of object types to the default standard output stream, with the default separator and end-of-line formatting added (these are the defaults because they are the most common use case):

```
>>> print()                                # Display a blank line

>>> x = 'python'
>>> y = 3.12
>>> z = ['lp6e']

>>> print(x, y, z)                          # Print objects per defaults
python 3.12 ['lp6e']
```

⁴ Technically, printing uses the *equivalent* of `str` in the internal implementation of Python, but the effect is the same. Besides this to-string conversion role, `str` is also the name of the string data type and can be used to decode Unicode strings from raw bytes with an extra encoding argument, as you’ll learn in [Chapter 37](#); this latter role is an advanced usage that you can safely ignore here.

There's no need to convert objects to strings here, as would be required for file write methods. By default, `print` calls add a space between the objects printed. To suppress this, send an empty string to the `sep` keyword argument, or send an alternative separator of your choosing:

```
>>> print(x, y, z, sep='')                                # Suppress separator
python3.12 ['lp6e']
>>>
>>> print(x, y, z, sep=', ')                              # Custom separator
python, 3.12, ['lp6e']
```

Also by default, `print` adds an end-of-line character to terminate the output line. You can suppress this and avoid the line break altogether by passing an empty string to the `end` keyword argument, or you can pass a different terminator of your own including a `\n` character to break the line manually if desired (the second of the following is two statements on one line, separated by a semicolon to demo the effect of custom terminators in the REPL):

```
>>> print(x, y, z, end='')                                # Suppress line break (see >>>)
python 3.12 ['lp6e']>>>
>>>
>>> print(x, y, z, end=''); print(x, y, z)                # Two prints, same output line
python 3.12 ['lp6e']python 3.12 ['lp6e']
>>> print(x, y, z, end='...\n')                          # Custom line end
python 3.12 ['lp6e']...
>>>
```

You can also combine keyword arguments to specify both separators and end-of-line strings—they may appear in any order but must appear after all the objects being printed:

```
>>> print(x, y, z, sep='...', end='!\n')                  # Multiple keywords
python...3.12...['lp6e']!
>>> print(x, y, z, end='!\n', sep='...')                  # Order doesn't matter
python...3.12...['lp6e']!
```

Here is how the `file` keyword argument is used—it directs the printed text to an open output file or other compatible object for the duration of the single `print` (this is really a form of stream redirection, a topic we will revisit later in this section):

```
>>> print(x, y, z, sep='...', file=open('data.txt', 'w'))  # Print to a file
>>> print(x, y, z)                                         # Back to stdout
python 3.12 ['lp6e']
>>> print(open('data.txt').read())                        # Display file text
python...3.12...['lp6e']
```

Finally, keep in mind that the separator and end-of-line options provided by `print` operations are just conveniences. If you need to display more specific formatting, don't print this way. Instead, build up a more custom string either ahead of time or within the `print` itself, using the string tools we mastered in [Chapter 7](#)—and print the string all at once. String *formatting* expressions and f-strings, for example, were designed for this sort of job:

```
>>> text = '%s: %-.4f, %05d' % (x, y, int(z[0][-2]))
>>> print(text)
python: 3.1200, 00006

>>> print(f'{x}: {y:-0.4f}, {int(z[0][-2]):05d}')
python: 3.1200, 00006
```

On the other hand, there seems to be a misconception that f-strings are somehow required for `print`, and they crop up needlessly. In truth, f-strings are overkill if you're simply trying to print objects separated by spaces:

```
>>> a, b, c, = 11, 3.14, 'hack'
>>> print(f'{a} {b} {c}')
11 3.14 hack
>>> print(a, b, c)
11 3.14 hack
```

As usual, don't use a sledgehammer to drive every nail!

Print Stream Redirection

As we've seen, `print` sends text to the standard output stream by default. However, it's often useful to send it elsewhere—to a text file, for example, to save results for later use or testing purposes. Although such redirection can usually be accomplished in system shells outside Python itself (with syntax like `> file`, per “[Command-Line Usage Variations](#)” on page 36), this is not a Python tool, and it's just as easy to redirect a script's streams from Python code.

The Python “hello world” program

Let's start off with the usual (and largely pointless) language benchmark—the “hello world” program. To print a “hello world” message in Python, simply print the string with `print`:

```
>>> print('hello world')           # Print a string object
hello world
```

Really, because expression results are echoed on the interactive command line, you often don't even need to use a `print` statement there—simply type the expressions you'd like to have printed, and their results are echoed back:

```
>>> 'hello world'                  # Interactive echoes
'hello world'
```

This code isn't exactly an earth-shattering feat of software mastery, but it serves to illustrate printing behavior. Technically, though, the `print` operation is just an *ergonomic* feature of Python—it provides a simple interface to the `sys.stdout` object, with a bit of default formatting. In fact, if you enjoy working harder than you must (or are pining for your Java coding days) you can also code print operations this way, which omits the `write` call's return value for space, as in [Chapter 9](#):

```
>>> import sys                     # Printing the hard way
>>> sys.stdout.write('hello world\n')
hello world
```

This code explicitly calls the `write` method of `sys.stdout`—an attribute preset when Python starts up to an open file object connected to the output stream. The `print` operation hides most of those details, providing a simple tool for simple printing tasks.

Manual stream redirection

So, why bother learning the hard way to print? The `sys.stdout` equivalent to `print` turns out to be the basis of a common technique in Python. In general, `print` and `sys.stdout` are directly related as follows. This statement:

```
print(X, Y)
```

is equivalent to the longer:

```
import sys
sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
```

which manually performs a string conversion with `str`, adds a separator and newline with `+`, and calls the output stream's `write` method. That is, this is what the `print` does. Which would you rather code?

Obviously, the long form isn't all that useful for printing by itself. However, it is useful to know that this is exactly what `print` operations do because it is possible to *reassign* `sys.stdout` to something different from the standard output stream. In other words, this equivalence provides a way of making your `print` operations send their text to other places. For example:

```
import sys
sys.stdout = open('log.txt', 'a')      # Redirects prints to a file
...
print(x, y, x)                        # Now printed text shows up in log.txt
```

Here, we reset `sys.stdout` to a manually opened file named *log.txt*, located in the script's working directory (CWD) and opened in `'a'` append mode (so we add to its current content). After the reset, every `print` operation anywhere in the program will write its text to the end of the file *log.txt* instead of to the original output stream. The `print` operations are happy to keep calling `sys.stdout`'s `write` method, no matter what `sys.stdout` happens to refer to. And because there is just one `sys` module in your program (technically, process), assigning `sys.stdout` this way will redirect every `print` anywhere in your program.

In fact, as the sidebar “[Why You Will Care: print and stdout](#)” on page 289 will explain, you can even reset `sys.stdout` to an object that isn't a file at all, as long as it has the expected interface: a method named `write` to receive the printed text-string argument. When that object is a *class*, printed text can be routed and processed arbitrarily per a `write` method you code yourself.

This trick of resetting the output stream might be more useful for programs originally coded with `print` statements. If you know that output should go to a file to begin with, you can always call file write methods instead. To redirect the output of a `print`-based program, though, resetting `sys.stdout` provides a convenient alternative to changing every `print` statement or using system shell-based redirection syntax, which may be above users' pay grades.

In other roles, streams may be reset to objects that display them in pop-up windows in GUIs, colorize them in IDEs like IDLE, and so on. It's a general technique.

Automatic stream redirection

Although redirecting printed text by assigning `sys.stdout` is a useful tool, a potential problem with the last section's code is that there is no direct way to restore the original output stream should you need to switch back after printing to a file. Because `sys.stdout` is just a normal file object, though, you can always save it and restore it if needed:⁵

```
>>> import sys
>>> temp = sys.stdout          # Save for restoring later
>>> sys.stdout = open('log.txt', 'a') # Redirect prints to a file

>>> print('lp6e was here')      # Prints go to file, not here
>>> print(1, 2, 3)
```

5 You may also be able to use the `__stdout__` attribute in the `sys` module, which refers to the original value `sys.stdout` had at program startup time. You still need to restore `sys.stdout` to `sys.__stdout__` to go back to this original stream value, though. See the `sys` module documentation for more details. Also note that `sys.stdout` and its cohorts may be `None` in some GUI programs with no console on which to display tests; be sure to check this where it matters.


```

>>> sys.stdout.close()                # Flush output to disk

>>> sys.stdout = temp                  # Restore original stream
>>> print('back in the REPL')          # Prints show up here again
back in the REPL
>>> print(open('log.txt').read())      # Result of earlier prints
lp6e was here
1 2 3

```

As you can see, though, manual saving and restoring of the original output stream like this involves extra juggling work. Because this crops up fairly often, a print extension is available to make it unnecessary.

As introduced earlier, the file keyword allows a single print call to send its text to the write method of a file (or file-like object), without actually resetting sys.stdout. Because the redirection is temporary, normal print calls keep printing to the original output stream. For example, the following abstract snippet again sends printed text to a file named *log.txt*:

```

log = open('log.txt', 'a')
print(x, y, z, file=log)           # Print to a file-like object
print(a, b, c)                     # Print to original stdout

```

These redirected forms of print are handy if you need to print to *both* files and the standard output stream in the same program. If you use these forms, however, be sure to give them a file object (or an object that has the same write method as a file object), not a file's name string. Here is the technique in action live:

```

>>> log = open('log2.txt', 'w')
>>> print(1, 2, 3, file=log)
>>> print(4, 5, 6, file=log)
>>> log.close()
>>> print(7, 8, 9)
7 8 9
>>> print(open('log2.txt').read())
1 2 3
4 5 6

```

These extended forms of print are also commonly used to print error messages to the standard *error* stream, available to your script as the preopened file object sys.stderr. You can either use its file write methods and format the output manually, or print with redirection syntax:

```

>>> import sys
>>> sys.stderr.write(('Bad!' * 8) + '\n')
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!

>>> print('Bad!' * 8, file=sys.stderr)
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!

```

Now that you know all about print redirections, the equivalence between printing and file write methods should be clear. The following demo prints both ways, then redirects the output to an external file to verify that the same text is printed (on Unix, you won't get the \r added to newlines here on Windows, and write results are back here):

```

>>> import sys
>>> X, Y = 1, 2

>>> print(X, Y)                                # Print: the easy way
1 2
>>> sys.stdout.write(str(X) + ' ' + str(Y) + '\n') # Print: the hard way

```

```

1 2
4

>>> print(X, Y, file=open('temp1', 'w'))           # Redirect text to file
>>> open('temp2', 'w').write(str(X) + ' ' + str(Y) + '\n') # Send to file manually
4

>>> print(open('temp1', 'rb').read())               # Binary mode for bytes
b'1 2\r\n'
>>> print(open('temp2', 'rb').read())
b'1 2\r\n'

```

As you can see, unless you happen to enjoy typing, print operations are often the best option for displaying text. For another example of the equivalence between prints and file writes, watch for a print function emulation example in [Chapter 18](#); it uses the coding patterns here to provide a print equivalent that can be customized.

Chapter Summary

In this chapter, we began our in-depth look at Python statements by exploring assignments, expressions, and print operations. Although these are generally simple to use, they have some alternative forms that, while optional, are often convenient in practice—for example, augmented and named assignments, as well as the redirection form of print operations, allow us to avoid some manual coding work. Along the way, we also studied the syntax of variable names, stream redirection techniques, and a variety of common mistakes to avoid, such as assigning the result of an append method call back to a variable.

In the next chapter, we'll continue our statement tour by filling in details about the if statement, Python's main selection tool; there, we'll also revisit Python's syntax model in more depth and look at the behavior of Boolean expressions, as well as the match multiple-choice statement. Before we move on, though, the end-of-chapter quiz will test your knowledge of what you've learned here.

Test Your Knowledge: Quiz

1. Name three ways that you can assign three variables to the same value.
2. What's dangerous about assigning three variables to a mutable object?
3. What's wrong with saying `L = L.sort()`?
4. How might you use the print operation to send text to an external file?

Test Your Knowledge: Answers

1. You can use multiple-target assignments (`A = B = C = 0`), sequence assignment (`A, B, C = 0, 0, 0`), or multiple assignment statements on three separate lines (`A = 0`, `B = 0`, and `C = 0`). With the latter technique, as introduced in [Chapter 10](#), you can also string the three separate statements together on the same line by separating them with semicolons (`A = 0; B = 0; C = 0`).
2. If you assign them this way: `A = B = C = []`, then all three names reference the same object, so changing it in place from one (e.g., `A.append(99)`) will affect the others. This is true only for in-place changes to mutable objects like lists and dictionaries; for immutable objects such as numbers and strings, this issue is irrelevant because they can never be changed in place.

3. The list `sort` method is like `append` in that it makes an in-place change to the subject list—it returns `None`, not the list it changes. The assignment back to `L` sets `L` to `None`, not to the sorted list. As discussed both earlier and later in this book (e.g., [Chapter 8](#)), a newer built-in function, `sorted`, sorts any sequence and returns a new list with the sorting result; because this is not an in-place change, its result can be safely and meaningfully assigned to a name.
4. To print to a file for a single print operation, you can use the `print(X, file=F)` call form, or assign `sys.stdout` to a manually opened file before the print and restore the original after if needed. You can also redirect all of a program's printed text to a file with special syntax in the system shell like `> file`, but this is outside Python's scope.

Why You Will Care: print and stdout

The equivalence between the `print` operation and writing to `sys.stdout` makes it possible to reassign `sys.stdout` to any user-defined object that provides the same `write` method as files. Because the `print` statement just sends text to the `sys.stdout.write` method, you can capture printed text in your programs by assigning `sys.stdout` to an object whose `write` method processes the text in arbitrary ways.

For instance, you can send printed text to a GUI window, or tee it off to multiple destinations, by defining an object with a `write` method that does the required routing. You'll see an example of this trick when we study classes in [Part VI](#) of this book, but as an abstract preview, it looks like this:

```
class FileFaker:
    def write(self, string):
        # Do something with the printed text in string

import sys
sys.stdout = FileFaker()
print(someObjects)           # Sends text to class write method
```

This works because `print` is what we will call in the next part of this book a *polymorphic* operation—it doesn't care what `sys.stdout` is, only that it has a method (i.e., interface) called `write`. This redirection to objects is made even simpler with the `file` keyword argument of `print`, because we don't need to reset `sys.stdout` explicitly—normal prints will still be routed to the `stdout` stream:

```
myobj = FileFaker()          # Redirect to object for one print
print(someObjects, file=myobj) # Does not reset sys.stdout
```

Python's built-in `input` function *reads* from the `sys.stdin` file, so you can intercept read requests in a similar way, using classes that implement file-like read methods instead. See the `input` and `while` loop example in [Chapter 10](#) for more background on this function.

Notice that because printed text goes to the `stdout` stream, it's also the way to print HTML reply pages in server-side scripts used on the web, and enables you to redirect Python script input and output at the operating system's shell command line as usual:

```
python script.py < inputfile > outputfile
python script.py | filterProgram
```

Python's `print` operation redirection tools are essentially pure-Python alternatives to some of these shell syntax forms. See other resources for more on web scripts and shell syntax.

if and match Selections

This chapter presents Python’s two statements used for selecting from alternative actions based on test results:

`if/elif/else`

The main selection workhorse in most programs, capable of coding arbitrary logic

`match/case`

A tool for narrower multiple-choice selection, with advanced matching operations

Because this is our first in-depth look at *compound statements*—statements that embed other statements—we will also explore the general concepts behind the Python statement syntax model here in more detail than we did in the introduction in [Chapter 10](#). Because the `if` statement introduces the notion of tests, this chapter will also deal with Boolean expressions, cover the “ternary” `if` expression, and fill in some details on truth tests in general.

if Statements

In simple terms, the Python `if` statement selects actions to perform. Along with its `if` expression counterpart, it’s the primary selection tool in Python and represents much of the *logic* a Python program possesses. It’s also our first compound statement. Like all such statements, the `if` statement may contain other statements, including other `ifs`. In fact, Python lets you combine statements in a program both sequentially (so that they execute one after another), and in an arbitrarily nested fashion (so that they execute only under certain conditions, such as selections and loops).

General Format

The Python `if` is typical of `if` statements in most procedural languages. It takes the form of an `if` test, followed by one or more optional `elif` (for “else if”) tests, and a final and optional `else` block. The tests and the `else` part each have an associated block of nested statements, indented under a header line. When the `if` statement runs, Python executes the block of code associated with the first test that evaluates to true, or the `else` block if all tests prove false. The general form of an `if` statement looks like this:

```

if test1:                # Main if test
    statements1          # Associated block
elif test2:              # Optional elif test(s)
    statements2          # Associated block
else:                    # Optional else default
    statements3          # Associated block

```

Basic Examples

To demonstrate, let's turn to a few simple examples of the `if` statement at work, in the REPL as usual. All parts are optional, except the initial `if` test and its associated statements. Thus, in the simplest case, the other parts are omitted:

```

$ python3
>>> if 1:
...     print('true')
...
true

```

As you've seen before, the prompt changes to `...` for continuation lines in many REPLs; in IDLE, you'll simply drop down to an indented line instead (and tap Backspace to back up when needed). A blank line, input by pressing Enter twice, terminates and runs the entire statement in most interactive interfaces.

Remember that `1` is Boolean `true` (as we'll review later, the word `True` is its equivalent), so this statement's test always succeeds. To handle a false result here, code the `else` to be run when the `if` test is not true:

```

>>> if not 1:
...     print('true')
... else:
...     print('false')
...
false

```

If you're working along: this chapter would like to omit all the `...` prompts for easier media copy and paste, but this would throw off indentation of associated lines like `else`. Instead, prompts are left off where possible, and code of some longer examples is listed without prompts, before its output. The book's examples package also has code sans prompts.

Here's a more complex `if` statement with all its optional parts present; it aims to display the roots of today's mobile operating systems (though it's not fully inclusive, and is prone to grow dated in a discriminating future near you):

```

if os in ['iOS', 'iPhoneOS']:
    print('macOS')
elif mode == 'mobile' and os != 'Windows':
    print('Linux')
else:
    print('unknown?')

>>> os, mode = 'Windows', 'mobile'
>>> ...insert the code above here...
unknown?

```

This multiline statement extends from the `if` line through the block nested under the `else`. When it's run, Python executes the statements nested under the first test that is true, or the `else` part if all tests are false (in this example, they are). In practice, both the `elif` and `else` parts may be omitted, and there

may be more than one statement nested in each section. Note that the words `if`, `elif`, and `else` are associated by the fact that they line up vertically, with the same indentation (ignoring the REPL prompts you may see if you paste and run this live).

The `and` expression in the preceding example is true if the expressions on its left and right sides are both true (more on such logical tests ahead). The `if` statement can also be *nested* to code choices that depend on others, and arbitrary logic in general. The following, for example, first ensures that it's dealing with a mobile before checking specific system names—logically speaking, there is an implied `and` between the enclosing and nested `ifs`:

```
if mode == 'mobile':
    if os == 'Android':
        print('Linux')
    elif os != 'Windows':
        print('macOS')

>>> os, mode = 'Android', 'mobile'
>>> ...insert the code above here...
Linux
```

Multiple-Choice Selections

Until Python's version 3.10, it had no *multiple-choice* selection statement similar to a “switch” in some other languages that selects an action based on a variable's value. As you'll learn ahead, Python today has sprouted a `match` statement that achieves the same goals (and substantially more!). Even so, multiple-choice logic can usually be coded just as easily by a series of `if/elif` tests and occasionally by indexing dictionaries or searching lists. Because dictionaries and lists can be built at runtime dynamically, they are often more flexible than hardcoded `if` (or `match`) logic in your script. The following, for instance, picks an operating system's release year, more or less, from its name:

```
>>> choice = 'Windows'
>>> print({'macos': 2001,           # A dictionary-based 'switch'
          'Linux': 1991,           # Use get() for a default (ahead)
          'Windows': 1985}[choice])

1985
```

Although it may take a few moments for this to sink in the first time you see it, this dictionary *is* a multiple-choice branch—indexing on the key `choice` branches to one of a set of values, much like a “switch” statement in other languages. An almost equivalent but more verbose Python `if` statement might look like the following:

```
if choice == 'macos':           # The equivalent if statement
    print(2001)
elif choice == 'Linux':
    print(1991)
elif choice == 'Windows':
    print(1985)
else:
    print('Bad choice')

>>> ...insert the code above here...
1985
```

Though it's perhaps more readable, the potential downside of an `if` like this is that, short of constructing it as a string and running it with tools like the `eval` or `exec` tools noted in [Chapter 10](#), it cannot

handle choices unknown until the program runs as easily as a dictionary. In more dynamic programs, data structures offer added flexibility.

Handling switch defaults

Notice the `else` clause on the `if` here to handle the default case when no key matches. As demoed in [Chapter 8](#), dictionary defaults can be coded with `in` expressions, `get` method calls, or exception catching with the `try` statement introduced in the preceding chapter. All of these same techniques can be used here to code a default action in a dictionary-based multiple choice. As a review in the context of this role, here's the `get` scheme at work with defaults (which also uses the more compact `dict` call to make the same dictionary):

```
>>> branch = dict(macos=2001, Linux=1991, Windows=1985)
>>> print(branch.get('Windows', 'Bad choice'))
1985
>>> print(branch.get('Solaris', 'Bad choice'))
Bad choice
```

An `in` membership test in an `if` statement can have the same default effect:

```
>>> choice = 'AmigaOS'
>>> if choice in branch:
...     print(branch[choice])
... else:
...     print('Bad choice')
...
Bad choice
```

And the `try` statement is a general way to handle dictionary-based defaults by catching and handling the errors they'd otherwise trigger (for more on exceptions, see [Chapter 11](#)'s overview and [Part VII](#)'s full treatment):

```
>>> choice = 'GEM'
>>> try:
...     print(branch[choice])
... except KeyError:
...     print('Bad choice')
...
Bad choice
```

Handling larger actions

As you can tell, dictionaries are good for associating values with keys, but what about the more complicated actions you can code in the statement blocks associated with `if` statements? In [Part IV](#), you'll learn that dictionaries can also contain *functions* to represent more complex actions and implement general “jump tables.” Such functions appear as dictionary values, may be coded as function names or inline `lambdas`, and are run by simply adding parentheses to trigger their actions.

Here's an abstract sample of this technique, but stay tuned for a rehash in [Chapter 19](#) after you've learned more about function definition:

```
def action(): ...
def default(): ...

branch = {'Android':    lambda: ...,      # A table of callable function objects
         'iOS':        action,           # Via inline lambdas, or def elsewhere
         'Symbian OS': lambda: ...}
```



```
choice = 'Android'
branch.get(choice, default)()           # Fetch and run associated action
```

Although dictionary-based multiple-choice branching is useful in programs that deal with more dynamic data, most programmers will probably find that coding an `if` statement is a straightforward way to perform this task. As a rule of thumb in coding, when in doubt, err on the side of simplicity and readability; it's the “Pythonic” way.

And the punch line here, of course, is that the `match` statement may handle *basic* multiple-choice selections better today—though it doesn't do general logic like `if`, can't handle dynamic data like dictionary indexing, and comes with substantial extra convolution for other roles. You'll have to move on to the next section to see all this for yourself.

match Statements

For most of Python's three-decade career, it resisted adding a multiple-choice selection statement, in large part because of all the options for coding such logic with existing tools that we just explored. Those options permeate vast amounts of Python code and remain relatively simple techniques that are perfectly valid to use in Python code written today.

Nevertheless, programming languages have a tendency to get caught up in *arms races* with each other—copying other languages' features and eroding their own distinctions in the process, and often for no better reason than familiarity with other tools. One of the fruits of this process is the `match` statement, new as of Python 3.10.

At its basic level, `match` is a potentially useful tool that adds a multiple-choice statement to Python. At this level, it works very much like “switch” statements in other languages and can be used instead of both `if/elif/else` combos and dictionary indexing in some contexts. In its full-blown form, however, `match` implements something known as *structural pattern matching*, which quickly falls off a complexity cliff, and seems a highly convoluted answer to a question that most Python programmers never asked. Especially for newcomers, it's a lot to justify.

Because of all that, this section is going to focus on the basic roles of `match`, and touch on its advanced pattern-matching roles only briefly, with delegation to Python's manuals for more of the story.

Basic match Usage

The good news is that `match`'s basic usage is straightforward. In its first-level form, `match` works the same as both an `if` statement and a dictionary index—like the following abstract snippets that emulate traffic lights in some locales (to run most examples here live, first assign variables to one of the options listed in their opening comments):

```
# state = 'go' or 'stop' or other

if state == 'go':
    print('green')
elif state == 'stop':
    print('red')
else:
    print('yellow')

# if-based multiple choice

colors = dict(go='green', stop='red')
print(colors.get(state, 'yellow'))

# Dictionary-based multiple choice
```

The equivalent `match` statement provides explicit syntax for such multiple-choice logic, at the cost of extra lines and extra indentation:

```
match state:
    case 'go':
        print('green')                # match-based multiple choice: 3.10+
    case 'stop':
        print('red')
    case _:
        print('yellow')
```

This statement works like this: `match` first evaluates the expression given in its header line (e.g., `state`) and then compares its result to values given in case header lines indented below it (e.g., `'go'`)—one after another, and top to bottom. As soon as a first match is found, the block of code nested under the matching case is run, and the entire `match` is exited. If no case values match, the `match` statement either runs the block under the case with value `_` (which provides a fallback *default*, and must appear last), or simply exits silently if no `_` case is present.

As usual, case blocks can contain multiple statements and nesting, and `match` itself can be nested in other statements. Moreover, case headers can also designate *multiple* values separated by `|` (“or”) which are all checked for a match, name a variable to be assigned to the matched value with `as`, and give a simple variable that is assigned the `match` expression’s result and always matches (and hence ends the statement, much like the anonymous `_`):

```
# state = 'go' or 'proceed' or 'start', or 'stop' or 'halt', or any other

match state:
    case 'go' | 'proceed' | 'start':    # Match any one of these 3
        print('green')                 # First left-to-right match wins
        print('means go')
    case 'stop' | 'halt' as what:       # Match any, and assign it to what
        print('red')                  # what outlives match if assigned
        print('means', what)
    case other:                         # Set other to state, and match
        print('catchall', other)       # other outlives match if assigned
```

In general, variables (like this example’s `what` and `other`) embedded in case headers and assigned during a successful match outlive the `match` statement itself: they can be used in code after the `match` exits, as long as that code is part of the same *scope*—which roughly means the same module or function, per coverage later in this book.

Match versus if live

As a live `match` example, the following maps statements to categories and assigns some variables along the way for later use; we’ll get formal about the `for` loop used here in the next chapter:

```
>>> for stmt in ['if', 'while', 'try']:
    match stmt:
        case 'if' | 'match':
            print('logic')
        case 'for' | 'while' as which:
            print('loop')
        case other:
            print('tbd')

logic
loop
tbd
```

```
>>> which, other
('while', 'try')
```

The equivalent `if` must assign the variables explicitly—though it requires less indentation, this example is artificial, and the `if` can handle more complex logic that the `match` cannot; it's not just a multiple-choice tool:

```
>>> for stmt in ['if', 'while', 'try']:
    if stmt in ['if', 'match']:
        print('logic')
    elif stmt in ['for', 'while']:
        which = stmt
        print('loop')
    else:
        other = stmt
        print('tbd')
```

...same results...

Watch for more basic `match` statements to show up later in this book (e.g., in Chapters 25, 30, and 38). Though it cannot be used to code general logic like `if`, `match` can make multiple-choice selection explicit in its basic form. Its extension to the *structural pattern matching* of the next section, however, is more difficult to rationalize—but you'll have to read on to judge for yourself.

Advanced match Usage

As noted, beyond its basic level shown so far, `match` becomes too complex to cover usefully here. In this guise, it goes well beyond multiple-choice logic, to define a language of its own for extracting components of structured objects. As a very brief survey, this over-cafeinated `match` treats case values as *patterns*, which may be:

- *Literal* patterns: a literal *X* matches the same value, by equality or identity
- *Wildcard* patterns: `_` matches anything, but the value is not assigned to `_`
- *Capture* patterns: variable *X* matches anything, and will be assigned to it
- *Or* patterns: *X* | *Y* | *Z* matches patterns *X* or *Y* or *Z*, stopping at the first match
- *As* patterns: *X* as *Y* matches pattern *X*, and assigns the matched value to *Y*
- Additional patterns that match *sequences*, *mappings*, *attributes*, and *instances* by structure

As an artificial (if frightening) example, [Example 12-1](#) demos *literal*, *sequence*, and *mapping* patterns. Its `[...]` and `(...)` patterns both match any sequence and are interchangeable; its `{...}` matches a mapping; and its single `*` or `**` names in patterns collect unmatched parts of a sequence or mapping, respectively. Using a `*` in this context is similar to [Chapter 11](#)'s extended-unpacking assignments, though here `**` is unique, not all parts of a pattern must be variables, and assignment to starred names is really a *side effect* of a Boolean test for a match (pasters beware: blank lines added for readability here won't work in a REPL, and `[...]` patterns preclude `(...)`s; run from a file and experiment freely).

Example 12-1. *matchdemo.py*

```
# state = 1 or
# [1, 2, 3] or [0, 2, 3] or (1, 2, 3) or (0, 2, 3) or
# dict(a=1, b=2, c=3) or dict(a=0, b=2, c=3) or other

match state:
    case 1 | 2 | 3 as what:          # Match integer literals, what = 1
        print('or', what)

    case [1, 2, what]:              # Match sequence (1), what = 3
        print('list', what)
    case [0, *what]:                # Match sequence (0), what = [2, 3]
        print('list', what)

    case {'a': 1, 'b': 2, 'c': what}: # Match mapping, what = 3
        print('dict', what)
    case {'a': 0, **what}:           # Match mapping, what = {'b': 2, 'c': 3}
        print('dict', what)

    case (1, 2, what):              # Match sequence: same as [1, 2, what]
        print('tuple', what)
    case (0, *what):                # Match sequence: same as [0, *what]
        print('tuple', what)

    case _ as what:                 # Match all other, what = other
        print('other', what)
```

Subtly, the [...], {...}, and (...) patterns in this code's case headers are not normal object literals. They're really *special-case syntax* forms that contain nested patterns, many of which just happen to be literal patterns here. Nested patterns can also be capture patterns (possibly after at most one * or **), and may use | ors, or _ wildcards. The case header [*a, 2 | 3, _], for example, is a valid sequence pattern, but has little to do with list literals.

Attribute and *instance* patterns require knowledge you haven't yet gained (a recurring theme in Python today), but they check for attribute values and inheritance-tree membership, and some of their components are nested patterns again, which may be literals, captures, and more. As a preview—which you can revisit after reading [Part VI](#):

```
class Emp:
    def __init__(self, name): self.name = name

pat = Emp('Pat')                      # pat.name becomes 'Pat': see Part VI!

# state = 'Pat' or pat

match state:
    case pat.name as what:           # Match object's attribute, what = 'Pat'
        print('attr', what)
    case Emp(name=what):             # Match an Emp instance, what = 'Pat'
        print('instance', what)
```

And if that's not already overkill for your multiple-choice logic needs, *parentheses* may be used around any pattern for grouping (as in general expressions); *nested* structures match recursively as they do in sequence assignment; and each case header can also end with an optional *guard* expression introduced by an if (after the optional as capture), which must be true for the case to be selected and its code block run:

```

state = ((1, 2), 3)
guard1 = True                                # a=1, b=3 if True; a=(1, 2) if False

match state:
    case ((a, 2), b) if guard1:                # Match+run only if guard1 is True
        print(f'case1 {a=} {b=}')
    case (a, 3) as what:                        # Reached only if guard1 is False
        print(f'case2 {a=} {what=}')
    case [a, (3 | 4)] as what if guard1:
        print(f'case3 {a=} {what=}')

```

All of which seems a blizzard of functionality to address usage that's overwhelmingly straightforward. Hence, this is where this book's `match` coverage must stop short for space (and mercy). For more on advanced roles of `match`, including all the gory details of its special-case patterns, consult Python's online resources.

Before you do, though, ponder just for a moment on the fact that Python was used successfully for three decades and rose to the top of the programming-languages heap *without* a `match` statement. Saddling the language with yet another convoluted subdomain that obviates a trivial amount of code might owe at least as much to hubris as user need.

That said, `match` may be useful in simpler roles, though you'll still need to choose between `if`, dictionaries fetches, and `match` when coding multiple-choice logic. While you should generally strive to avoid doing something just because you've done it in other languages (you're now using Python, after all), such choices are yours to make.

Python Syntax Revisited

Python's syntax model was introduced in [Chapter 10](#). Now that we're stepping up to larger statements like `if` and `match`, this section reviews and expands on the syntax ideas introduced earlier. In general, Python has a simple, statement-based syntax. Among its highlights:

- **Statements execute one after another, until you say otherwise.** Python normally runs statements in a file or nested block in order from first to last, but statements like `if` (as well as loops and exceptions) cause the interpreter to jump around in your code. Because Python's path through a program is called the control flow, statements such as `if` that affect it are often called *control-flow statements*.
- **Block and statement boundaries are detected automatically.** As we've seen, there are no braces or "begin/end" delimiters around blocks of code in Python; instead, Python uses the indentation of statements under a header to group the statements in a nested block. Similarly, Python statements are not normally terminated with semicolons; rather, the end of a line usually marks the end of the statement coded on that line. As special cases you'll meet later, statements can both span lines and be combined on a line when it's useful.
- **Compound statements = header + ":" + indented statements.** All Python *compound statements*—those with nested statements—follow the same pattern: a header line terminated with a colon, followed by one or more nested statements, usually indented under the header. The indented statements are called a *block* (or sometimes, a *suite*). In the `if` statement, the `elif` and `else` are part of the `if`, but they are also header lines with nested blocks of their own. As a special case, blocks can be on the same line as the header if they are not compound.

- **Blank lines, spaces, and comments are usually ignored.** Blank lines are both optional and ignored in files (but not at the interactive prompt, when they terminate compound statements). Spaces inside statements and expressions are almost always ignored (except in string literals, and when used for indentation). Comments are always ignored: they start with a `#` character (not inside a string literal) and extend to the end of the current line.
- **Docstrings are ignored but are saved and displayed by tools.** Python supports an additional comment form called documentation strings (*docstrings* for short), which, unlike `#` comments, are retained at runtime for inspection. Docstrings are simply strings that show up at the top of program files and some statements. Their content is ignored, but they are attached to objects and may be displayed with tools covered later in this book.

For most Python newcomers, the lack of the braces and semicolons used to mark blocks and statements in many other languages seems to be the most novel syntactic feature of Python, so let's explore what this means in more depth.

Block Delimiters: Indentation Rules

As introduced in [Chapter 10](#), Python detects block boundaries automatically, by line *indentation*—that is, the empty space to the left of your code. This section is a rehash of the rules, with a few more details sprinkled in along the way.

In short, all statements indented the same distance to the right belong to the same block of code. In other words, the statements within a block line up vertically, as in a column. The block ends when a lesser-indented line or the end of the file is encountered (or you enter a blank line in a REPL), and more deeply nested blocks are simply indented further to the right than the statements in the enclosing block.

For instance, [Figure 12-1](#) demonstrates the block structure of the following code:

```
x = 1
if x:
    y = 2
    if y:
        print('block2')
    print('block1')
print('block0')
```

This code contains three blocks: the first (the top-level code of the file) is not indented at all, the second (within the outer `if` statement) is indented four spaces, and the third (the `print` statement under the nested `if`) is indented eight spaces.

Top-level (unnested) code must start in column 1, but nested blocks can start in any column; indentation may consist of any number of spaces and tabs, as long as it's the same for all the statements in a given single block. That is, Python doesn't care *how* you indent your code; it only cares that it's done consistently. Four spaces or one tab per indentation level are common conventions, but there is no absolute standard or rule in the Python world.

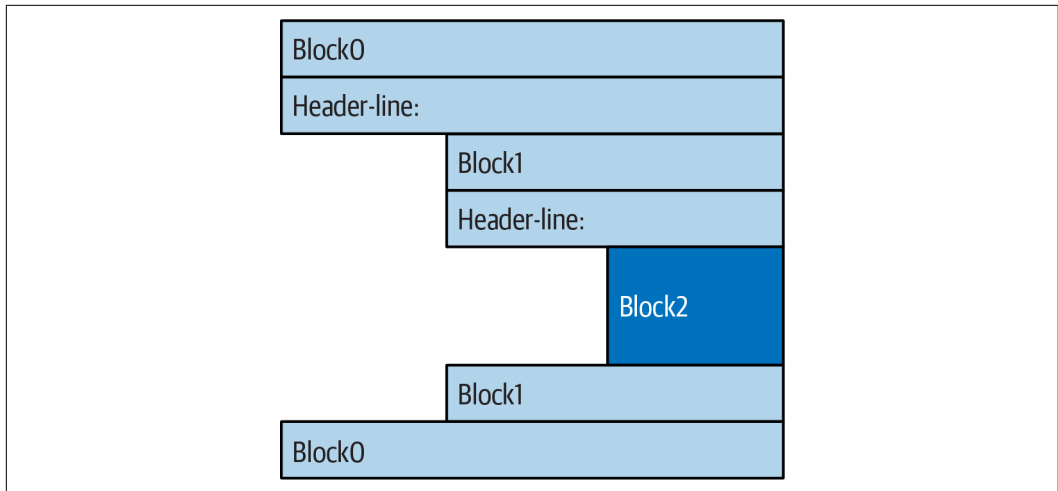


Figure 12-1. Nested blocks of code denoted by their indentation

Indenting code is quite natural in practice. For example, the following fully frivolous code snippet demonstrates common indentation errors in Python code, which are easy to spot because they’re visually askew:

```
x = 'Hack'                                # Error: first line indented
if 'tho' in 'python':
    print(x * 8)
    x += 'More!'                          # Error: unexpected indentation
    if x.endswith('re!'):
        x *= 2
    print(x)                             # Error: inconsistent indentation
```

The properly indented version of this code looks like the following—even for an artificial example like this, proper indentation makes the code’s intent much more apparent:

```
x = 'Hack'
if 'tho' in 'python':
    print(x * 8)                          # Prints 8 Hack
    x += 'More!'
    if x.endswith('re!'):
        x *= 2
    print(x)                             # Prints HackMore!HackMore!
```

It’s important to know that the only major place in Python where whitespace matters is where it’s used to the *left* of your code, for indentation; in most other contexts, space can be coded or not. However, indentation is really part of Python syntax, not just a stylistic suggestion: all the statements within any given single block must be indented to the same level, or Python reports a syntax error. This is intentional—because you don’t need to explicitly mark the start and end of a nested block of code, some of the syntactic clutter found in other languages is unnecessary in Python.

As described in [Chapter 10](#), making indentation part of the syntax model also enforces consistency, a crucial component of readability in structured programming languages like Python. In Python’s syntax, the indentation of each line of code unambiguously tells readers what it is associated with. This uniform and consistent appearance in turn makes Python code easier to maintain and reuse.

In the end, indentation is easier than you may think. Consistently indented code always satisfies Python's rules, and most text editors (including IDLE) make it easy to follow Python's model by automatically indenting as you type.

Avoid mixing tabs and spaces

That said, there's one rule of thumb you should know: although you can use spaces or tabs to indent, it's usually not a good idea to *mix* the two within a block—use one or the other. Technically, tabs count for enough spaces to move the current column number up to a multiple of 8, and your code will work if you mix tabs and spaces consistently. However, mixing tabs and spaces makes code difficult to read and change completely apart from Python's syntax rules—tabs may look very different in the next programmer's editor than they do in yours.

For these reasons, Python issues an error when a script mixes tabs and spaces for indentation inconsistently within a block (that is, in a way that makes it dependent on a tab's equivalent in spaces). So don't do that: when in Python do as Pythoners do and use consistent indentation instead of block delimiters.

Statement Delimiters: Lines and Continuations

While blocks are indented, a statement in Python normally ends at the end of the line on which it appears. When a statement is too long to fit on a single line, though, a few special rules may be used to make it span multiple lines:

- **Statements may span multiple lines if you're continuing an "open pair."** The code of a statement can always be continued on the next line if it's enclosed in a `()`, `{}`, or `[]` pair. For instance, expressions in parentheses and dictionary and list literals can span any number of lines; the statement doesn't end until the end of the line containing the closing part of the pair (a `)`, `}`, or `]`). *Continuation lines*—lines 2 and beyond of the statement—can start at any indentation level, but it's best to align vertically for readability in some fashion.
- **Statements may span multiple lines if they end in a backslash.** Though best used as a fallback option, if a statement needs to span multiple lines, you can also add a backslash—a `\` not embedded in a string literal or comment—at the end of the prior line to indicate you're continuing on the next line. Because you can also continue by adding parentheses around most constructs, backslashes are rarely used today. This approach is also error-prone: accidentally forgetting a `\` may generate a syntax error or cause the next line to run independently.
- **Statements may be combined if separated with a semicolon.** Though uncommon, you can terminate a statement with a semicolon. This is sometimes used to squeeze more than one statement onto a single line by separating them with semicolons, but this works only when the combined statements are not compound.
- **Statements may contain multiline strings.** As we learned in [Chapter 7](#), triple-quoted string blocks are designed to span multiple lines normally. We also learned in [Chapter 7](#) that adjacent string literals are implicitly concatenated; when this is used in conjunction with the open-pairs rule mentioned earlier, wrapping this construct in parentheses allows it to span multiple lines.

Special Syntax Cases in Action

Here's what a continuation line looks like using the open-pairs rule just described. Delimited constructs, such as lists in square brackets, can span across any number of lines:

```
L = ['app',  
    'script',  
    'program']           # Open pairs may span lines
```

This also works for list comprehensions enclosed in `[]`; anything in `()` (tuples, expressions, function argument and headers, and generators expressions); and anything in `{}` (dictionary and set literals and comprehensions). Some of these are tools we'll study in later chapters, but this rule naturally covers most constructs that span lines in practice.

If you're accustomed to using backslashes to continue lines, you can in Python, too, but it's not common practice:

```
if a == b and c == d and \  
    d == e and f == g:  
    print('old school')    # Backslashes allow continuations...
```

Because any expression can be enclosed in parentheses, you can usually use the open-pairs technique instead if you need your code to span multiple lines—simply wrap a part of your statement in parentheses:

```
if (a == b and c == d and  
    d == e and e == f):  
    print('new school')    # But parentheses usually do too, and are obvious
```

In fact, backslashes are generally discouraged, because they're too easy to not notice and too easy to omit altogether. In the following, `x` is assigned `10` with the backslash, as intended; if the backslash is accidentally omitted, though, `x` is assigned `6` instead, and *no error is reported* (the `+4` is a valid expression statement by itself). In a real program with a more complex assignment, this could be the source of a very obscure bug:

```
x = 1 + 2 + 3 \           # Omitting the \ makes this very different!  
+4
```

As another special case, Python allows you to write more than one noncompound statement (i.e., statements without nested statements) on the same line, separated by semicolons. Some coders use this form to save program file real estate, but it usually makes for more readable code if you stick to one statement per line for most of your work:

```
x = 1; y = 2; print(x)    # More than one simple statement
```

As covered in [Chapter 7](#), triple-quoted string literals span lines too. In addition, if two string literals appear next to each other, they are concatenated as if a `+` had been added between them—when used in conjunction with the open-pairs rule, wrapping in parentheses allows this form to span multiple lines. For example, the first of the following inserts newline characters at line breaks and assigns `S` to `'\naaaa\nbbbb\ncccc'`, and the other two implicitly concatenate and assign `S` to `'aaaabbbbcccc'`; as we also saw in [Chapter 7](#), `#` comments are ignored in the second form but *included* in the string in the first, and *f-strings* require `f` prefixes even on continuations lines:

```

S = """
aaaa
bbbb
cccc"""

S = ('aaaa'
     'bbbb'          # Comments here are ignored, add \n if needed
     'cccc')

S = (f{'a' * 4}'      # Also makes 'aaaabbbbcccc'
     f{'b' * 4}'      # Use f'' on each f-string part
     r'cccc')         # And ditto for r'' raw-string parts

```

Finally, and also as a review, Python lets you move a compound statement's body up to the header line, provided the body contains just simple (noncompound) statements. You'll most often see this used for simple `if` statements with a single test and action, as in the interactive loops we coded in [Chapter 10](#):

```

if 1: print('hello')          # Simple statement on header line

```

With a little effort, you can combine some of these special cases to write Python code that is difficult to read, but it's not generally recommended. As a rule of thumb, try to keep each statement on a line of its own, and indent all but the simplest of blocks. Six months down the road, you'll be happy you did.

Truth Values Revisited

The notions of comparison, equality, and truth values were introduced in [Chapter 9](#). Like syntax, though, the `if` is the first statement we've studied that actually uses these tools, so we'll rehash these ideas with additional info. All told, Python's Boolean operators are a bit different from their counterparts in some other languages. In Python:

- All objects have an inherent Boolean true or false value.
- Any nonzero number or nonempty object is true.
- Zero numbers, empty objects, and the special object `None` are considered false.
- Comparisons and equality tests are applied recursively to data structures.
- Comparisons and equality tests return `True` or `False` (custom versions of 1 and 0).
- Boolean `and` and `or` operators return a true or false operand *object*.
- Boolean operators stop evaluating (*short-circuit*) as soon as a result is known.

The `if` statement takes action on truth values, but *Boolean* operators are used to combine the results of other tests in richer ways to produce new truth values. More formally, there are three Boolean expression operators in Python:

```

X and Y
    Is true if both X and Y are true—and returns either X or Y

X or Y
    Is true if either X or Y is true—and returns either X or Y

not X
    Is true if X is false—and returns either True or False

```

Here, *X* and *Y* may be any truth value, or any expression that returns a truth value (e.g., an `==` equality test, an `in` membership check, and so on). Boolean operators are typed out as words in Python (instead of C's `&&`, `||`, and `!`), and shouldn't be confused with Python's `&`, `|`, and `^` operators that work on numbers and sets (and dictionaries).

Most uniquely, Boolean `and` and `or` operators return a true or false *object* in Python, not the values `True` or `False`. Let's turn to a few examples to see how this works:

```
>>> 2 < 3, 3 < 2      # Less than: return True or False (1 or 0)
(True, False)
```

Magnitude comparisons such as these return `True` or `False` as their truth results, which, as we learned in Chapters 5 and 9, are really just custom versions of the integers 1 and 0 (they print themselves differently but are otherwise the same).

Boolean operators `and` and `or`, on the other hand, always return an object—either the object on the *left* side of the operator or the object on the *right*. If we test their results in `if` or other statements, they will be as expected (remember, every object is inherently true or false), but we won't get back a simple `True` or `False`.

For `or` tests, Python evaluates the operand objects from left to right and returns the first one that is *true*. Moreover, Python stops at the first true operand it finds. This is usually called *short-circuit evaluation*, as determining a result short-circuits (terminates) the rest of the expression as soon as the result is known:

```
>>> 2 or 3, 3 or 2      # Return left operand if true
(2, 3)                 # Else, return right operand (true or false)
>>> [] or 3
3
>>> [] or {}
{}
```

In the first line of the preceding example, both operands (2 and 3) are true (i.e., are nonzero), so Python always stops and returns the one on the left—which determines the result because true or anything is always true. In the other two tests, the left operand is false (an empty object), so Python simply evaluates and returns the object on the right—which may have either a true or a false value when tested, but determines the result of the `or` at large.

Python `and` operations also stop (short-circuit) as soon as the result is known. However, in this case Python evaluates the operands from left to right and stops if the left operand is a *false* object because it determines the result—false and anything is always false:

```
>>> 2 and 3, 3 and 2    # Return left operand if false
(3, 2)                 # Else, return right operand (true or false)
>>> [] and {}
[]
>>> 3 and []
[]
```

Here, both operands are true in the first line, so Python evaluates both sides and returns the object on the right—which determines the result of the `and`. In the second test, the left operand is false (`[]`), so Python stops and returns it as the `and` result without ever running the code on the right side. In the last test, the left side is true (3), so Python evaluates and returns the object on the right—which happens to be a false `[]`.

The net effect of all this apparent nonsense is the same as in most other languages—you get a value that is logically true or false if tested in an `if` or `while` according to the normal definitions of `or` and `and`. However, in Python, Booleans return either the left or the right *object*, not a simple integer flag.

This behavior of `and` and `or` may seem esoteric at first glance, but see this chapter’s sidebar, “[Why You Will Care: Booleans](#)” on page 308, for examples of how it is sometimes used to advantage in Python coding. The next section also shows a common way to leverage this behavior, and its more mnemonic alternative.

The if/else Ternary Expression

One common role for the prior section’s Boolean operators is to code an expression that runs the same as an `if` statement. To get started, consider the following very common code, which sets `A` to either `Y` or `Z`, based on the truth value of `X`:

```
if X:
    A = Y
else:
    A = Z
```

Sometimes, though, the items involved in such a statement are so simple that it seems like overkill to spread them across four lines. At other times, we may want to nest such a construct in a larger statement instead of assigning its result to a variable separately. For such reasons (and possibly to appease ex-C programmers), Python includes a “ternary” (three-part) expression that allows us to say the same thing in just one line of code:

```
A = Y if X else Z
```

This expression has the exact same effect as the preceding four-line `if` statement, but it’s simpler to code. In some sense, it is to `if` statements what the prior chapter’s `:=` is to assignment statements: an expression equivalent with more limited syntax and narrower roles, which is nevertheless useful in some code. For example, you can’t code full statements in the parts of this expression, but you can embed it anywhere that Python allows an expression.

As in the statement equivalent, the ternary expression runs expression `Y` only if `X` turns out to be true and runs expression `Z` only if `X` turns out to be false. That is, it *short-circuits*, just like the Boolean operators described in the prior section, running just `Y` or `Z` but not both. Here are some examples of it in action:

```
>>> tone = 'formal'
>>> a = 'code' if tone == 'formal' else 'hack'
>>> a
'code'

>>> tone = 'informal'
>>> a = 'code' if tone == 'formal' else 'hack'
>>> a
'hack'
```

The same effect can often be achieved by a careful combination of `and` and `or` operators, because they return either the object on the left side or the object on the right as the preceding section described. The ternary expression in the following works the same as the Boolean expression below it:

```
A = Y if X else Z           # Ternary if/else

A = ((X and Y) or Z)       # and+or equivalent
```

This works, but there is a catch—you have to be able to assume that `Y` will be Boolean true. If that is the case, the effect is the same: the `and` runs first and returns `Y` if `X` is true; if `X` is false the `and` skips `Y` and returns false `X`, and the `or` simply returns `Z`. In other words, we get “if `X` then `Y` else `Z`”—which is exactly what the ternary expression says, albeit in a different order.

The `and/or` combination form also seems to require a “moment of great clarity” to understand the first time you see it, which qualifies as an argument against its deployment. As a guideline: use the equivalent and more robust and mnemonic `if/else` expression when you need this structure, or use a full `if` statement when the parts are nontrivial.

As a side note (and just in case this section hasn’t made your head explode yet), using the following expression is similar to the prior Boolean `and` ternary expressions, because the `bool` function will translate any `X` into the equivalent of integer 1 or 0 (i.e., `True` or `False`), which can then be used as an offset to pick true and false values from a list:

```
A = [Z, Y][bool(X)]
```

Truth be told, the `bool` is not required when `X` already yields a truth value, but is when `X` is an object like a string:

```
>>> ['hack', 'code'][tone == 'formal']
'hack'
>>> ['hack', 'code'][bool('formal')]
'code'
```

But alas, this isn’t exactly the same, because Python will not *short-circuit*—it will always run both `Z` and `Y`, regardless of the value of `X`. Because of such complexities, you’re better off using the simpler and more easily digested `if/else` expression. Even then, common sense goes a long way here as always. Like most nestable tools, the ternary expression is naturally prone to yield code that’s tough to read. If you find yourself working hard at packing logic into one, consider taking a moment to think about how hard it will be to unpack later. Your coworkers will be glad you did.

Chapter Summary

In this chapter, we studied the Python `if` statement. Additionally, because this was our first compound and logical statement, we reviewed Python’s general syntax rules and explored the operation of truth values and tests in more depth than we were able to previously. Along the way, we also looked at how to code multiple-choice logic in Python with both dictionaries and `match`, learned about the `if/else` ternary expression, and explored Boolean operators.

The next chapter continues our look at procedural statements by expanding the coverage of `while` and `for` loops. There, you’ll learn about alternative ways to code loops in Python, some of which may be better than others. Before that, though, here is the usual chapter quiz to review before moving ahead.

Test Your Knowledge: Quiz

1. How might you code a multiple-choice branch in Python?
2. How can you code an `if/else` statement as an expression in Python?
3. How can you make a single statement span many lines?
4. What do the words `True` and `False` mean?
5. What does “short-circuiting” mean, and where does it crop up?

Test Your Knowledge: Answers

1. An `if` statement with multiple `elif` clauses is often the most straightforward way to code a multiple-choice branch, though not necessarily the most concise or flexible. Dictionary indexing can often achieve the same result, especially if the dictionary contains callable functions coded with `def` statements or `lambda` expressions. As of Python 3.10, the `match` statement provides explicit syntax for multiple-choice selections; it works well in basic roles but can't code logic as general as that in the `if` statement and comes with substantial heft in support of structural pattern matching, a very different tool.
2. The expression form `Y if X else Z` returns `Y` if `X` is true, or `Z` otherwise; it's the same as a four-line `if` statement and works well in limited contexts, but can't code actions as rich as those in the full `if` statement, and has the potential to produce code that's hard to read. The `and/or` combination `((X and Y) or Z)` can work the same way, but it's more obscure and requires that the `Y` part be true.
3. Wrap up the statement in an open syntactic pair `((, [, or {`), and it can span as many lines as you like; the statement ends when Python sees the closing (right) half of the pair, and lines 2 and beyond of the statement can begin at any indentation level. Backslash continuations work, too, but are broadly discouraged in the Python world.
4. This is partly a review from [Chapter 9](#), but is reinforced in the sidebar at the end of this chapter. `True` and `False` are just custom versions of the integers `1` and `0`, respectively: they always stand for Boolean true and false values in Python. They're available for use in truth tests and variable initialization and are printed for expression results at the interactive prompt. In all these roles, they serve as a more mnemonic and hence readable alternative to `1` and `0`.
5. Short-circuiting happens when Python stops evaluating an expression early because its result can already be determined from the expression so far. It comes up in `and` and `or` expressions, which run their right side only if their left sides don't determine their results. It also comes up in the `if/else` ternary expression, which runs either its true or false parts, depending on its test part's logical result.

Why You Will Care: Booleans

One common way to use the somewhat unusual behavior of Python Boolean operators is to select from a set of objects with an `or`. A statement such as this:

```
X = A or B or C or None
```

assigns `X` to the first nonempty (that is, true) object among `A`, `B`, and `C`, or to `None` if all of them are empty. This works because the `or` operator returns one of its two objects, and it turns out to be a fairly common coding paradigm in Python: to select a nonempty object from among a fixed-size set, simply string them together in an `or` expression. In simpler form, this is also commonly used to designate a default—the following sets `X` to `A` if `A` is true (nonzero or nonempty), and to default otherwise:

```
X = A or default
```

It's also important to understand the short-circuit evaluation of Boolean operators and the `if/else`, because it may prevent actions from running. Expressions on the right of a Boolean operator, for example, might call functions that perform substantial or important work, or have side effects that won't happen if the short-circuit rule takes effect:

```
if f1() or f2(): ...
```

Here, if `f1` returns a true (or nonempty) value, Python will never run `f2`. To guarantee that both functions will be run, call them before the `or`:

```
tmp1, tmp2 = f1(), f2()
if tmp1 or tmp2: ...
```

You've already seen another application of this behavior in this chapter: because of the way Booleans work, the expression `((A and B) or C)` can be used to emulate an `if` statement—almost (see this chapter's discussion of this form for details).

We met additional Boolean use cases in prior chapters. As we saw in [Chapter 9](#), because all objects are inherently true or false, it's common and easier in Python to test an object directly (`if X:`) than to compare it to an empty value (`if X != ''`). For a string, the two tests are equivalent. As we also saw in [Chapter 5](#), the preset Boolean values `True` and `False` are the same as the integers `1` and `0` and are useful for initializing variables (`X = False`), for loop tests (`while True:`), and for displaying results at the interactive prompt.

Also watch for related discussion in operator overloading in [Part VI](#): when we define new object types with classes, we can specify their Boolean nature with either the `__bool__` or `__len__` methods. The latter of these is tried if the former is absent and designates false by returning a length of zero—because an empty object is considered false.

Finally, and as a preview, other tools in Python have roles similar to the `or` chains at the start of this sidebar: the `filter` call and list comprehensions you'll explore later can be used to select true values when the set of candidates isn't known until runtime (though they evaluate all values and return all that are true), and the `any` and `all` built-ins can be used to test if any or all items in a collection are true (they short-circuit their checks like `and`, `or`, and `if/else`, but don't select an item per se):

```
>>> L = [1, 0, 2, 0, 'hack', '', 'py', []]
>>> list(filter(bool, L))           # Get true values
[1, 2, 'hack', 'py']
>>> [x for x in L if x]             # Comprehensions
[1, 2, 'hack', 'py']
>>> any(L), all(L)                 # Aggregate truth
(True, False)
```

As we've learned, the `bool` function here simply returns its argument's true or false value, as though it were tested in an `if`. Watch for more on these related tools in [Chapters 14](#), [19](#), and [20](#).

while and for Loops

This chapter concludes our tour of Python procedural statements by presenting the language’s two main *looping* constructs—statements that repeat an action over and over:

while/else

The most general looping statement, which can handle repetitive tasks of all kinds

for/else

A specialized loop designed for stepping through the items in any “iterable” object easily

We’ve met and used both of these informally already, but we’ll fill in additional usage details here. While we’re at it, we’ll also study a few less prominent statements used within loops, such as `break` and `continue`, the `loop else`, and cover some built-ins commonly used with loops, such as `range`, `zip`, and `enumerate`.

Although the `while` and `for` statements covered here are the primary syntax provided for coding repeated actions, there are additional looping operations and concepts in Python. Because of that, the iteration story is continued in the next chapter, where we’ll explore the related ideas of Python’s *iteration protocol* (used by the `for` loop) and *list comprehensions* (a close cousin to the `for` loop). Later chapters explore even more exotic iteration tools such as *generators* and functional tools like `map`, `filter`, and `reduce`. For now, though, let’s keep things simple.

while Loops

Python’s `while` statement is the most general iteration construct in the language. In simple terms, it repeatedly executes an associated block of statements as long as a test at the top keeps evaluating to a true value. It is called a “loop” because control keeps looping back to the start of the statement until the test becomes false. When the test does become false, control passes to the statement that follows the `while` block. The net effect is that the loop’s body is executed repeatedly while the test at the top is true. If the test is false to begin with, the body never runs and the `while` statement is skipped.

General Format

In its most complex form, the `while` statement consists of a header line with a test expression, a body of one or more normally indented statements, and an optional `else` part that is executed if control exits

the loop without a `break` statement being encountered. Python keeps evaluating the test at the top and executing the statements nested in the loop body until the test returns a false value:

```
while test:           # Loop test
    statements        # Repeated loop body
else:                 # Optional else
    statements         # Run if didn't exit loop body with break
```

Examples

To illustrate, let's look at a few simple `while` loops in action. The first, which consists of a `print` statement nested in a `while` loop, just prints a message forever. Recall that `True` is just a custom version of the integer `1` and always stands for a Boolean true value; because the test is always true, Python keeps executing the body forever or until you stop its execution. This sort of behavior is usually called an *infinite loop*—it's not really immortal, but you may need a `Ctrl+C` key combination to forcibly terminate it:

```
>>> while True:
...     print('Type Ctrl+C to stop me!')
```

The next example keeps *slicing* off the first character of a string until the string is empty and hence false (and begins omitting the REPL's `...` prompts for easier media copy and paste where possible). It's typical to test an object directly like this instead of using the more verbose equivalent (`while x != ''`), though later in this chapter you'll see other ways to step through the items in a string more easily with a `for` loop:

```
>>> x = 'code'
>>> while x:
    print(x, end=' ')
    x = x[1:]
```

While x is not empty
Print next character
Strip first character off x

code ode de e

Note the `end=' '` keyword argument used here to place all outputs on the same line separated by a space; see [Chapter 11](#) if you've forgotten why this works as it does. This will probably leave your REPL's input prompt at the end of the output's line; type `Enter` (or your keyboard's or app's equivalent) to reset if desired.

The following code *counts* from the value of `a` up to, but not including, `b`. Loops like this are often used to generate object indexes. You'll also see an easier way to do this with a Python `for` loop and the built-in `range` function later:

```
>>> a=0; b=10
>>> while a < b:
    print(a, end=' ')
    a += 1
```

One way to code counter loops
Or, a = a + 1

0 1 2 3 4 5 6 7 8 9

Finally, notice that Python doesn't have what some languages call a “do until” loop statement. However, we can simulate one with a test and `break` at the bottom of the loop body, so that the loop's body is always run at least once:

```
while True:
    ...loop body...
    if test: break
```

Always run loop body at least once
Test for loop exit at the bottom

To fully understand how this structure works, we need to move on to the next section's coverage of `break`.

break, continue, pass, and the Loop else

Now that we've seen a few Python loops in action, it's time to take a look at two simple statements that have a purpose only when nested inside loops—the `break` and `continue` statements. While we're looking at oddballs, we will also study the `loop else` clause here because it is intertwined with `break`, as well as Python's empty placeholder statement `pass`, which is not tied to loops but falls into the category of simple one-word statements. In Python:

break

Jumps out of the closest enclosing loop (past the entire loop statement)

continue

Jumps to the top of the closest enclosing loop (to the loop's header line)

pass

Does nothing at all: it's an empty statement placeholder

Loop else block

Runs if and only if the loop is exited normally (i.e., without hitting a `break`)

General Loop Format

Factoring in `break` and `continue` statements, the general format of the `while` loop looks like this:

```
while test:
    statements
    if test: break           # Exit loop now, skip else if present
    if test: continue       # Go to test at top of loop now
else:
    statements              # Run on exit if didn't hit a 'break'
```

`break` and `continue` statements can appear anywhere inside the `while` (or `per ahead`, `for`) loop's body, but they are usually coded further nested in an `if` test to take action in response to some condition.

Let's turn to a few simple examples to see how these statements come together in practice.

pass

Simple things first: the `pass` statement is a no-operation placeholder that is coded when the syntax requires a statement, but you have nothing useful to say. It is often used to code an empty body for a compound statement. For instance, if you want to code an infinite loop that does nothing each time through, do it with a `pass`:

```
while True: pass           # Type Ctrl+C to stop me!
```

Because the body is just an empty statement, Python gets stuck in this loop. `pass` is roughly to statements as `None` is to objects—an explicit nothing. Notice that here the `while` loop's body is on the same line as the header, after the colon; as with `if` statements, this works only if the body isn't a compound statement (and doesn't contain one).

This example does nothing forever. It probably isn't the most useful Python program ever written (unless you want to warm up your laptop or phone on a cold winter's day), but it's tough to come up with a better `pass` example at this point in the book; it's not a commonly used tool.

You'll see other places where `pass` makes a bit more sense later—for instance, to ignore exceptions caught by `try` statements and to define empty `class` objects with attributes that behave like “structs” and “records” in other languages. Though partly a preview, a `pass` is also sometime coded to mean “to be filled in later,” to stub out the bodies of functions temporarily:

```
def func1():
    pass                # Add real code here later

def func2():
    pass
```

We can't leave the body empty without getting a syntax error, so we say `pass` instead.

The ellipsis-literal alternative

Despite the limited roles, there's a similar, if more obscure, way to achieve the same effect as `pass`. Python allows an *ellipsis*, coded as `...` (literally, three consecutive dots, not the Unicode character), to appear any place an expression can. Because ellipses do nothing by themselves, they can serve as an alternative to the `pass` statement, especially for code to be filled in later—a sort of Python TBD:

```
def func1():
    ...                # Alternative to pass

func1()                # Does nothing if called
```

This works because any expression can appear as a statement (as we learned in [Chapter 11](#)), and the `...` literal qualifies as an expression. Ellipses can also appear on a statement header by itself, and may be used to initialize variable names if no specific type is required—which also makes it an alternative to `None`:

```
def func1(): ...      # Works on same line too

>>> tbd = ...         # Alternative to both pass and None
>>> tbd               # Ellipsis is a real (if oddball!) thing
Ellipsis
```

This goes well beyond the original intent of `...` in slicing extensions (which, like the `@` operator and type hinting, is unused by Python itself), so time will tell if it rises to challenge `pass` and `None` in these inane and vacuous roles.

continue

The `continue` statement causes an immediate jump to the top of a loop. It's often used to avoid statement nesting, as in the next example that uses `continue` to skip odd numbers. This code prints all even numbers less than 10 and greater than or equal to 0. Recall that 0 means false and `%` is the remainder-of-division (modulus) operator, so this loop counts down to 0, skipping numbers that aren't multiples of 2—and prints 8 6 4 2 0:

```
x = 10
while x:
    x -= 1                # Or, x = x - 1
    if x % 2 != 0: continue # Odd? -- skip print
    print(x, end=' ')

```

Because `continue` jumps to the top of the loop, you don't need to nest the `print` statement here inside an `if` test; the `print` is only reached if the `continue` is not run.

The nested-code alternative

If all this sounds similar to a “go to” in other languages, it should. Python has no “go to” statement, but because `continue` lets you jump about in a program, many of the warnings about readability and maintainability you may have heard about “go to” apply. `continue` should probably be used sparingly, especially when you're first getting started with coding. For instance, the last example might be clearer if the `print` were nested under the `if`:

```
x = 10
while x:
    x -= 1
    if x % 2 == 0:                # Even? -- print
        print(x, end=' ')
```

Later in this book, you'll also learn that raised and caught exceptions can also emulate “go to” statements in limited and structured ways; stay tuned for more on this technique in [Chapter 36](#) where you will learn how to use it to break out of multiple nested loops, a feat not possible with the next section's topic alone.

break

The `break` statement causes an immediate exit from a loop—technically, from the closest enclosing loop, when loops are nested. Because the code that follows it in the loop is not executed if the `break` is reached, it can sometimes avoid nesting much like `continue`. For example, here is a simple interactive loop (a takeoff on code we studied in [Chapter 10](#)) that inputs data with `input` and exits when the user enters a “stop” line:

```
>>> num = 1
>>> while True:
    tool = input(f'{num}) What\'s your favorite language? ')
    if tool == 'stop': break
    print('Bravo!' if tool == 'Python' else 'Try again...')
    num += 1

1) What's your favorite language? Java
Try again...
2) What's your favorite language? Python
Bravo!
3) What's your favorite language? stop
```

Because the `break` in this terminates the `while` immediately, there's no reason to nest code below it in an `else`.

The named-assignment alternative

That said, it's also possible to use the newer `:=` expression we met in “[Named Assignment Expressions](#)” on [page 274](#) to crunch this example—albeit, at the expense of its role as a `break` demo. While you should judge for yourself, the net effect is concise but may at least flirt with unreadability:

```
>>> num = 1
>>> while (tool := input(f'{num}) What\'s your favorite language? ')) != 'stop':
    print('Bravo!' if tool == 'Python' else 'Try again...')
    num += 1
```

```
1) What's your favorite language? Python
Bravo!
```

Nesting `:=` within `:=` as in the following, however, could easily incite pitchforks and torches (in fact, the full one-liner here is too wide for this book!). Unless you can defend this in a court of your code-reuse peers, just say no:

```
num = 0
while (tool := input(f'{(num := num + 1)}) What\'s your favorite language? ')) != 'stop':
```

Preview: in [Chapter 36](#), you'll see that `input` also raises an exception at end-of-file (e.g., if the user enters Ctrl+Z on Windows or Ctrl+D on Unix); wrapping `input` in `try` statements allows users to respond this way too.

Loop else

When combined with the `loop else` clause, the `break` statement can often eliminate the need for the search status flags used in other languages. In abstract terms the `break` in the following skips the `else` on the way out of the loop:

```
while continuing:
    if found:
        found code
        break
    else advance
else:
    not-found code
```

As a more concrete example, the following piece of code determines whether a positive integer `num` is prime—has no factors other than 1 and itself—by searching for factors greater than 1 (to run live, assign `num` before pasting):

```
x = num // 2                                # For some num > 1, start at half
while x > 1:
    if num % x == 0:                        # Remainder 0? Factor found
        print(num, 'has factor', x)
        break                               # Exit now and skip else
    x -= 1
else:                                       # Normal exit, when x reaches 1
    print(num, 'is prime')
```

Rather than setting a flag to be tested when the loop is exited, it inserts a `break` where a factor is found. This way, the `loop else` clause can assume that it will be executed only if no factor is found; if this code never hits the `break`, the number is prime. Trace through this code to see how this works.

The `loop else` clause is also run if the body of the loop is *never* executed, as you don't run a `break` in that event either; in a `while` loop, this happens if the test in the header is false to begin with. Thus, in the preceding example you still get the “is prime” message if `x` is initially less than or equal to 1 (for instance, if `num` is 2).



Subprime code: This example determines primes, but only informally so. Numbers less than 2 are not considered prime by the strict mathematical definition, but 1 and 0 are classified as such here. To be really picky, this code also fails for negative numbers and succeeds for floating-point numbers with all-zero decimal digits. Also note that its code must use `//` instead of `/` because we need the initial division to truncate remainders, not retain them. If you want to experiment with this code further, watch for its associated exercise at the end of [Part IV](#), which wraps it in a function for reuse.

Why the loop else?

Because the loop `else` clause is unique to Python, it tends to perplex some newcomers (and even some veterans; in fact, a few either pointlessly code the loop `else` without a `break` or don't know that the loop `else` exists at all!). In general terms, the loop `else` simply provides explicit syntax for a common coding scenario—it is a coding structure that lets us catch the “other” way out of a loop, without setting and checking flags or conditions.

Suppose, for instance, that we are writing a loop to search a list for a value and need to know whether the value was found after you exit the loop. We might code such a task this way (this code is intentionally abstract and incomplete; `x` is a sequence and `match` is a tester function to be defined):

```
found = False
while x and not found:
    if match(x[0]):
        print('Found')
        found = True
    else:
        x = x[1:]
if not found:
    print('Not found')
```

Value at front?
Slice off front and repeat

Here, we initialize, set, and later test a `found` flag to determine whether the search succeeded or not. This is valid Python code, and it does work; however, this is exactly the sort of structure that the loop `else` clause is meant to handle. Here's an `else` equivalent:

```
while x:
    if match(x[0]):
        print('Found')
        break
    x = x[1:]
else:
    print('Not found')
```

Exit when x empty
Exit, go around else
Only here if exhausted x

This version is more concise. The flag is gone, and we've replaced the `if` test at the loop end with an `else` (lined up vertically with the word `while`). Because the `break` inside the main part of the `while` exits the loop and goes around the `else`, this serves as a more structured way to catch the search-failure case.

Some readers might have noticed that the prior example's `else` clause could be replaced with a test for an empty `x` after the loop (e.g., `if not x:`). Although that's true in this example, the `else` provides explicit syntax for this coding pattern (it's more obviously a search-failure clause here), and such an explicit empty test may not apply in some cases. The loop `else` becomes even more useful when used in conjunction with the `for` loop—the topic of the next section—because sequence iteration is not under your control.

for Loops

The for loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable object. All told, the for statement works on strings, lists, tuples, sets, dictionaries, and all other built-in iterables, as well as new user-defined objects that you'll learn how to create later with classes. We met for briefly in [Chapter 4](#) and have used it in conjunction with sequence object types; let's expand on its usage more formally here.

General Format

The Python for loop begins with a header line that specifies an assignment target (or targets), along with the object you want to step through. The header is followed by a block of (normally indented) statements that you want to repeat:

```
for target in object:      # Assign object items to target
    statements             # Repeated loop body: use target
else:                     # Optional else
    statements             # Run if didn't exit loop body with break
```

When Python runs a for loop, it assigns the items in the iterable *object* to the *target* one by one and executes the loop body for each. The loop body typically uses the assignment target to refer to the current item in the sequence as though it were a cursor stepping through the sequence.

While *target* can be any assignment target we met in [Chapter 11](#), it's often just a simple name. This name is a possibly new variable that lives in the scope where the for statement itself is coded. There's not much unique about this name; it can even be changed inside the loop's body, but it will automatically be set to the next item in *object* when control returns to the top of the loop again. After the loop this variable normally still refers to the last item visited, which is the last item in the sequence unless the loop exits early with a break statement.

The for statement also supports an optional else block, which works exactly as it does in a while loop—it's executed if the loop exits without running into a break statement (i.e., if all items in the sequence have been visited). The break and continue statements introduced earlier also work the same in a for loop as they do in a while. Given all that, the for loop's complete format can be described this way:

```
for target in object:      # Assign object items to target
    statements
    if test: break         # Exit loop now, skip else
    if test: continue      # Go to top of loop now
else:
    statements             # Run on exit if didn't hit a 'break'
```

Examples

Let's type a few for loops interactively now, so you can see how they are used in practice.

Basic usage

As mentioned earlier, a for loop can step across any kind of sequence object. In our first example, for instance, we'll assign the name *x* to each of the three items in a list in turn, from left to right, and the print statement will be executed for each. Inside the print statement (the loop body), the name *x* refers to the current item in the list:


```
>>> for x in ['app', 'script', 'program']:
    print(x, end=' ')
```

```
app script program
```

The next two examples compute the sum and product of all the items in a list. Later in this chapter and later in this book you'll meet tools that apply operations such as `+` and `*` to items in a list automatically, but it's often just as easy to use a `for`:

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
    sum = sum + x

>>> sum
10
>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod *= item

>>> prod
24
```

Other data types

Any sequence works in a `for`, as it's a generic tool. For example, `for` loops also work on strings and tuples:

```
>>> S = 'Python'
>>> T = ('web', 'num', 'app')

>>> for x in S: print(x, end=' ')    # Iterate over a string

P y t h o n

>>> for x in T: print(x, end=' ')    # Iterate over a tuple

web num app
```

In fact, as we'll explore in the next chapter when we formalize the notion of iterables, `for` loops can even work on some objects that are not sequences—including files.

Tuple (sequence) assignment in `for` loops

If you're iterating through a sequence of tuples, the loop target itself can actually be a *tuple* of targets. This is just another case of the tuple-unpacking assignment we studied in [Chapter 11](#) at work. Remember, the `for` loop *assigns* items in the sequence object to the target, and assignment works the same everywhere:

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:                # Tuple assignment at work
    print(a, b)

1 2
3 4
5 6
```

Here, the first time through the loop is like running $(a,b) = (1,2)$, the second time is like $(a,b) = (3,4)$, and so on. The net effect is to automatically *unpack* the current tuple on each iteration. List syntax works as a for target too because tuple and list assignment are both *sequence* assignment, and tuple parentheses are optional:

```
>>> for [a, b] in T:           # List assignment: same effect
>>> for a, b in T:             # Tuple sans parentheses: same effect
```

This list-of-tuples data format is commonly used in conjunction with the `zip` call you'll meet later in this chapter, to implement parallel traversals. It also crops up in conjunction with SQL databases in Python, where query result tables are returned as sequences of sequences like the list used here—the outer list is the database table, the nested tuples are the rows within the table, and tuple (i.e., sequence) assignment extracts columns.

As we've seen in earlier chapters, tuples in for loops also come in handy to iterate through *both* keys and values in dictionaries using the `items` method, rather than looping through the keys and indexing to fetch the values manually:

```
>>> D = {'a': 1, 'b': 2}
>>> for key in D:
    print(key, '=>', D[key])           # Use dict keys iterator and index

a => 1
b => 2

>>> list(D.items())
[('a', 1), ('b', 2)]

>>> for (key, value) in D.items():
    print(key, '=>', value)           # Iterate over both keys and values

a => 1
b => 2
```

It's important to note that tuple assignment in for loops isn't a special case; *any* assignment target works syntactically after the word `for`. For example, we can always assign manually within the loop to unpack:

```
>>> T
[(1, 2), (3, 4), (5, 6)]

>>> for both in T:
    a, b = both                       # Manual assignment equivalent
    print(a, b)

1 2
3 4
5 6
```

But tuples in the loop header save us an extra step when iterating through sequences of sequences. As suggested in [Chapter 11](#), even *nested* structures may be automatically unpacked this way in a for:

```
>>> ((a, b), c) = ((1, 2), 3)        # Nested sequences work too
>>> a, b, c
(1, 2, 3)

>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: print(a, b, c)

1 2 3
4 5 6
```

Even this is not a special case, though—the `for` loop simply runs the sort of assignment we ran just before it, on each iteration. Any nested sequence structure may be unpacked this way, simply because *sequence assignment* is so generic:

```
>>> for (a, b, c) in [(1, 2), 3], ['XY', 6]: print(a, b, c)

1 2 3
X Y 6
```

Extended-unpacking assignment in `for` loops

In fact, because the loop variable in a `for` loop can be *any* assignment target, we can also use the starred names and other targets of extended-unpacking assignment here to extract both items and sections of sequences within sequences. Because this works in assignment statements, it automatically works in `for` loops too.

This topic was introduced in [Chapter 11](#), but here's a quick refresher to reinforce the technique. Consider the tuple assignment form introduced in the prior section. A tuple of values is assigned to a tuple of names on each iteration, exactly like a simple assignment statement:

```
>>> a, b, c = (1, 2, 3)                                # Tuple assignment
>>> a, b, c
(1, 2, 3)

>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:            # Used in for loop
    print(a, b, c)

1 2 3
4 5 6
```

Because sequence assignment supports a more general set of names with a starred target to collect multiple items, we can use the same syntax to extract parts of nested sequences in the `for` loop:

```
>>> a, *b, c = (1, 2, 3, 4)                            # Extended-unpacking assignment
>>> a, b, c
(1, [2, 3], 4)

>>> for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
    print(a, b, c)

1 [2, 3] 4
5 [6, 7] 8
```

In practice, this approach might be used to pick out multiple columns from rows of data represented as nested sequences. As usual in Python, you can achieve similar effects with more basic tools—in this case by slicing. The only difference is that slicing returns a type-specific result, whereas starred targets always receive lists:

```
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:            # Manual slicing version
    a, b, c = all[0], all[1:-1], all[-1]
    print(a, b, c)

1 (2, 3) 4
5 (6, 7) 8
```

Finally, all the starred-target forms work in `for` as in = assignment statements, including nested sequences, indexes, and slices (though practical roles for code like the following are probably much more rare than common!):

```
>>> L, M = [1, 2], [3, 4]
>>> pairs = [[(5, 6), (7, 8), (9, 10)]] * 2

>>> for [(a, *X), (b, *L[0]), (c, *M[:0])] in pairs:
    print(f'<{a=} {X=}> <{b=} {L=}> <{c=} {M=}>')

<a=5 X=[6]> <b=7 L=[[8], 2]> <c=9 M=[10, 3, 4]>
<a=5 X=[6]> <b=7 L=[[8], 2]> <c=9 M=[10, 10, 3, 4]>
```

See [Chapter 11](#) for more on the extended-unpacking form of assignment.

Nested for loops

Now let's look at some for loops that are a bit more sophisticated than those demoed so far. The first shows what happens when for loops are nested—the inner loop is run for every iteration of the outer loop, and the + within the inner loop combines their items by using each loop's variable:

```
>>> for x in 'abc':
    for y in '123':
        print(x + y, end=' ')
# For each item in one string
# And for each item in another string
# Concatenate current items from both

a1 a2 a3 b1 b2 b3 c1 c2 c3
```

The next example kicks the nesting up a notch, illustrating both three-level statement nesting and the loop `else` clause in a `for`. Given a list of objects (`items`) and a list of keys (`tests`), this code searches for each key in the objects list and reports on the search's outcome:

```
>>> items = ['aaa', 111, (4, 5), 2.01]
>>> tests = [(4, 5), 3.14]
>>>
>>> for key in tests:
    for item in items:
        if item == key:
            print(key, 'was found')
            break
    else:
        print(key, 'not found!')

(4, 5) was found
3.14 not found!
```

Because the nested `if` runs a `break` when a match is found, the inner loop's `else` clause can assume that if it is reached, the search has failed. Notice the nesting here. When this code runs, there are two loops going at the same time: the outer loop scans the keys list, and the inner loop scans the items list for each key. The nesting of the loop `else` clause is critical; it's indented to the same level as the header line of the inner `for` loop, so it's associated with the inner loop, not the `if` or the outer `for`.

The preceding example is illustrative, but it may be easier to code if we employ the `in` operator to test membership. Because `in` implicitly scans an object looking for a match (at least logically), it replaces the inner loop:

```
>>> for key in tests:
    if key in items:
        print(key, 'was found')
    else:
        print(key, 'not found!')
# For all keys
# Let Python check for a match

(4, 5) was found
3.14 not found!
```

In general, it's a good idea to let Python do as much of the work as possible (as in this solution) for the sake of both brevity and performance.

Our final example is similar, but builds a list as it goes for later use instead of printing. It performs a typical data-structure task with a `for`—collecting common items in two sequences (it's nearly *intersection*, unless there are duplicate values). After the loop runs, `res` refers to a list that contains all the items found in `seq1` and `seq2`:

```
>>> seq1 = 'trippy'
>>> seq2 = 'python'
>>>
>>> res = []                                # Start empty
>>> for x in seq1:                          # Scan first sequence
>>>     if x in seq2:                      # Common item?
>>>         res.append(x)                 # Add to result end

>>> res
['t', 'p', 'p', 'y']
```

Unfortunately, this code is equipped to work only on two specific variables: `seq1` and `seq2`. It would be nice if this loop could somehow be generalized into a tool you could use more than once. As you'll see, that simple idea leads us to *functions*, the topic of the next part of the book.

Of course, if you read [Chapter 4](#) or [Chapter 5](#), you know that Python has sets that provide true intersection with the `&` operator—but the result's order is scrambled, duplicates are dropped, and multiple conversions are required to match:

```
>>> list(set(seq1) & set(seq2))            # Real intersection with sets
['p', 'y', 't']
```

More usefully, this code also exhibits the classic *list comprehension* pattern—collecting a results list with an iteration and optional filter test—and could be coded much more concisely with this tool:

```
>>> [x for x in seq1 if x in seq2]         # Let Python collect results
['t', 'p', 'p', 'y']
```

But you'll have to read on to the next chapter for the rest of this story.

Loop Coding Techniques

The `for` loop we just studied subsumes most counter-style loops. It's generally simpler to code and often quicker to run than a `while`, so it's the first tool you should reach for whenever you need to step through a sequence or other iterable. In fact, as a general rule, you should *resist the temptation to count things in Python*—its iteration tools automate much of the work you do to loop over collections in lower-level languages like C.

Still, there are situations where you will need to iterate in more specialized ways. For example, what if you need to visit every second or third item in a list, or change the list along the way? How about traversing more than one sequence in parallel, in the same `for` loop? What if you need indexes too?

You can always code such unique iterations with a `while` loop and manual indexing, but Python provides a set of built-ins that allow you to specialize the iteration in a `for`:

- The built-in `range` function produces a series of successively higher integers, which can be used as indexes in a `for`.

- The built-in `zip` function returns a series of parallel-item tuples, which can be used to traverse multiple sequences in a `for`.
- The built-in `enumerate` function generates both the values and indexes of items in an iterable, so we don't need to count manually.

Because `for` loops may run quicker than `while`-based counter loops, it's to your advantage to use tools like these that allow you to use `for` whenever possible. Let's look at each of these built-ins in turn, in the context of common roles. As you'll see, some loop-coding alternatives are more valid than others.

Counter Loops: `range`

Our first loop-related function, `range`, is a general tool that can be used in a variety of contexts. We met it briefly in [Chapter 4](#) and have used it occasionally along the way. Although it's used often to generate indexes in a loop, you can call it anywhere you need a series of integers (see [Chapter 11](#)'s enumerated-names trick for a prime example).

As we've seen, `range` is an *iterable* that generates items on demand, so we need to wrap it in a `list` call to display all its results at once in a REPL. Surprisingly, `range`'s results support *some* sequence operations, but not all; per [Chapter 9](#), it was reclassified in Python's docs as a sort of sequence object type, though one with less functionality than lists and tuples—and much less basis in reality:

```
>>> list(range(5)), list(range(2, 5)), list(range(0, 10, 2))
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])

>>> range(5)[2], range(5)[1:3], list(range(5)) + [6, 7]
(2, range(1, 3), [0, 1, 2, 3, 4, 6, 7])

>>> range(5) + [6, 7]
TypeError: unsupported operand type(s) for +: 'range' and 'list'
```

Categorization aside, `range` usage is straightforward. With one argument, `range` generates a series of integers from zero up to *but not including* the argument's value. If you pass in two arguments, the first is taken as the *lower* bound. And an optional third argument can give a *step*; if it is used, Python adds the step to each successive integer in the result (the step defaults to +1). Ranges can also be nonpositive and nonascending, if you need them to be:

```
>>> list(range(-5, 5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(range(5, -5, -1))
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

We'll take a deeper look at iterables in [Chapter 14](#). In this case, Python 2.X had an optimized built-in named `xrange`, which was like its `range` but didn't build a result list in memory all at once; which was later superseded in 3.X by the generator behavior of its `range`; which was later rebranded a sequence by 3.X docs (confusingly!). The upshot of this long walk is that today's `range` doesn't consume much space, because it produces numbers only on demand.

Although the preceding `range` results may be useful all by themselves, they tend to come in most handy within `for` loops. For one thing, they provide a simple way to repeat an action a specific number of times. To print three lines, for example, use a `range` to generate the appropriate number of integers:

```
>>> for i in range(3):
    print(i, 'Pythons')

0 Pythons
1 Pythons
2 Pythons
```

Note that `for` loops force results from `range` automatically, so we don't need to use a list wrapper here. In fact, we *shouldn't*: letting `range` produce its results one at a time uses much less memory than forcing them all at once.

Sequence Scans: while, range, and for

The `range` call is also sometimes used to iterate over a sequence indirectly, though it's often not the best approach in this role. The easiest and fastest way to step through a sequence exhaustively is almost always with a simple `for`, because Python handles most of the details for you in quick, internal code:

```
>>> X = 'hack'
>>> for item in X: print(item, end=' ')      # Automatic iteration with for

h a c k
```

Internally, the `for` loop handles the details of the iteration automatically when used this way. If you really need to take over the indexing logic explicitly (and sometimes you may), you can do it with a `while` loop:

```
>>> i = 0
>>> while i < len(X):
    print(X[i], end=' ')
    i += 1                                # Manual iteration with while

h a c k

>>> i = -1
>>> while (i := i + 1) < len(X):
    print(X[i], end=' ')                  # Manual, but with := operator

h a c k
```

You can also do manual indexing with a `for`, though, if you use `range` to generate indexes to iterate through. It's a multistep process—you must ask for the `range` of the subject's `len`—but it's sufficient to generate offsets, rather than the items at those offsets:

```
>>> X
'hack'
>>> len(X)                                # Length of string
4
>>> list(range(len(X)))                    # All legal offsets into X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): print(X[i], end=' ') # Manual range/len iteration

h a c k
```

Importantly, because this example is stepping over a list of *offsets* into `X`, not the actual *items* of `X`, we need to index back into `X` within the loop to fetch each item. If this seems like overkill, though, it's because it is: there's really no reason to work this hard in this example.

Although the `range/len` combination is useful in some roles, it's probably not the best option in most. It may run slower, and it's also more code than we need to write. Unless you have a special indexing requirement, you're better off using the simple `for` loop form in Python:

```
>>> for item in X: print(item, end=' ')          # Use auto iteration if you can
```

As guidelines, use `for` instead of `while` whenever possible, and don't use `range` calls in `for` loops except as a last resort. This simpler solution is almost always better. Like every good guideline, though, there are plenty of exceptions—as the next section demonstrates.

Sequence Shufflers: `range` and `len`

Though not ideal for simple sequence scans, the `range/len` coding pattern used in the prior example does allow us to do more specialized sorts of traversals when required. For example, some algorithms can make use of sequence *reordering*—to generate alternatives in searches, to test the effect of different value orderings, and so on. Such cases may require offsets in order to pull sequences apart and put them back together, as in the following; its `range`'s integers provide a repeat count in the first, and a position for slicing in the second:

```
>>> S = 'hack'
>>> for i in range(len(S)):      # For repeat counts 0..3
    S = S[1:] + S[:1]           # Move front item to end
    print(S, end=' ')
```

ackh ckha khac hack

```
>>> S
'hack'
>>> for i in range(len(S)):      # For positions 0..3
    X = S[i:] + S[:i]           # Rear part + front part
    print(X, end=' ')
```

hack ackh ckha khac

Trace through these one iteration at a time if they seem confusing. The second creates the same results as the first, though in a different order, and doesn't change the original variable as it goes. Because both slice to obtain parts to concatenate, they also work on any type of sequence, and return sequences of the same type as that being shuffled—if you shuffle a list, you create reordered lists:

```
>>> L = [1, 2, 3, 4]
>>> for i in range(len(L)):
    X = L[i:] + L[:i]           # Works on any sequence type
    print(X, end=' ')
```

[1, 2, 3, 4] [2, 3, 4, 1] [3, 4, 1, 2] [4, 1, 2, 3]

The results of `range` itself, however, don't make the grade in either coding (they're not true sequences!):

```
>>> L = range(4)
>>> ...same code as prior example...
TypeError: unsupported operand type(s) for +: 'range' and 'range'
```

We'll make use of code like this to test functions with different argument orderings in [Chapter 18](#), and will extend it to functions, generators, and more complete permutations in [Chapter 20](#)—it's a widely useful tool.

Skipping Items: range and Slices

The prior section showed one valid applications for the range/len combination. We might also use this technique to *skip* items as we go:

```
>>> S = 'abcdefghijk'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print(S[i], end=' ')

a c e g i k
```

Here, we visit every *second* item in the string S by stepping over the generated range list. To visit every *third* item, change the third range argument to be 3, and so on. In effect, using range this way lets you skip items in loops while still retaining the simplicity of the for statement.

In many or most cases, though, this is also probably not the “best practice” technique in Python today. If you really mean to skip items in a sequence, the extended three-limit form of the *slice expression*, presented in [Chapter 7](#), provides a simpler route to the same goal. To visit every second character in S, for example, slice with a stride of 2:

```
>>> S = 'abcdefghijk'
>>> for c in S[::2]: print(c, end=' ')

a c e g i k
```

The result is the same, but substantially easier for you to write and for others to read. The potential advantage to using range here instead is space: slicing makes a copy of the string, while range does not—and hence may save significant memory for very large strings. Naturally, whether your program needs to care depends on what it does.

Changing Lists: range and Comprehensions

Another common place where you may use the range/len combination with for is in loops that *change* a list as it is being traversed. Suppose, for example, that you need to add 1 to every item in a list (maybe you’re updating ages at year end). You can try this with a simple for loop, but the result may not be what you want or expect:

```
>>> L = [10, 20, 30, 40, 50]

>>> for x in L:
    x += 1                                # Changes x, not L!

>>> L                                     # L's objects unchanged
[10, 20, 30, 40, 50]

>>> x                                     # x is not a cursor into L
51
```

This doesn’t quite work—it changes the loop variable x, not the list L. The reason is somewhat subtle. Each time through the loop, x refers to the next integer already pulled out of the list. In the first iteration, for example, x is integer 10, taken from L. When we then add to x in the loop body with +=, it sets x to a different object, integer 11, but it does not update the list where 10 originally came from; the new 11 is a piece of memory separate from the list.

To really change the list as we march across it, we need to use indexes so we can assign an updated value to each position as we go. The `range/len` combination can produce the required indexes for us:

```
>>> L = [10, 20, 30, 40, 50]

>>> for i in range(len(L)):      # Add one to each item in L
    L[i] += 1                    # Or L[i] = L[i] + 1

>>> L
[11, 21, 31, 41, 51]
```

When coded this way, the list is changed as we proceed through the loop. There is no way to do the same with a simple `for x in L`, because such a loop iterates through actual *items*, not their positions. But what about the equivalent `while` loop? Such a loop requires a bit more work on our part, and might run more slowly depending on your Python, your host device, and perhaps the alignment of planets (you'll see how to check such claims in [Chapter 21](#)):

```
>>> i = 0
>>> while i < len(L):           # And similar with := assignment
    L[i] += 1
    i += 1

>>> L
[12, 22, 32, 42, 52]
```

Here again, though, the `range` solution may not be ideal either. A list *comprehension* expression of the form:

```
>>> [x + 1 for x in L]
[13, 23, 33, 43, 53]
```

likely runs faster today and would do similar work, albeit without changing the original list in place (we could assign the expression's new list object result back to `L`, but this would not update any other references to the original list). Because this is such a central looping concept, we'll save a complete exploration of list comprehensions for the next chapter, and tell the rest of the statements story of loops there.

Parallel Traversals: `zip`

Our next loop coding technique adds to its bag of tricks. As we've seen, the `range` built-in allows us to traverse sequences with `for` in a nonexhaustive fashion. In a similar spirit, the built-in `zip` function allows us to use `for` loops to visit multiple sequences *in parallel*—not overlapping in time, but during the same loop. In basic operation, `zip` takes one or more arguments (sequences or other iterables) and returns a series of tuples that pair up parallel items taken from those arguments. For example, suppose we're working with two lists of data paired by position:

```
>>> L1 = [1, 2, 3, 4]
>>> L2 = [5, 6, 7, 8]
```

To combine the items in these lists, we can use `zip` to create a list of tuple pairs. Like `range`, `zip` is an *iterable* object, so we must wrap it in a `list` call to collect and display all its results at once (again, the next chapter will be more formal about iterables like this):

```
>>> zip(L1, L2)                  # An iterable that generates pairs
<zip object at 0x026523C8>
>>> list(zip(L1, L2))           # list() required to see all results
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Such a result may be useful in other contexts as well, but when wedded with the for loop, it supports parallel iterations:

```
>>> for (x, y) in zip(L1, L2):
    print(f'{x} + {y} => {x + y}')

1 + 5 => 6
2 + 6 => 8
3 + 7 => 10
4 + 8 => 12
```

Here, we step over the result of the `zip` call—that is, the pairs of items pulled from the two lists. Notice that this for loop again uses the tuple (a.k.a. sequence) assignment form we met earlier to unpack each tuple in the `zip` result. The first time through, it's as though we ran the assignment statement `(x, y) = (1, 5)`; and so on.

The net effect is that we scan both `L1` and `L2` in our loop. To be sure, we could achieve a similar effect with a `while` loop that handles indexing manually—like the following that produces the same output as the preceding:

```
>>> i = -1
>>> while (i := i + 1) < len(L1):
    print(f'{L1[i]} + {L2[i]} => {L1[i] + L2[i]}')
```

But this requires noticeably more code, and hence would likely run slower than the `for/zip` approach. Moreover, it's no better on space: being an iterable, `zip` makes just one pair per loop, and so does not consume memory needlessly. The clincher, though, is that this is not really equivalent to `zip`—for reasons disclosed in the next section.

More on zip: size and truncation

For the record, the `zip` function is more general than the prior example suggests. For instance, it both *is* an iterable and *accepts* any type of iterable object, including range results, input files, and more:

```
>>> list(zip(range(4), 'hack'))
[(0, 'h'), (1, 'a'), (2, 'c'), (3, 'k')]
```

In addition, `zip` is not just for two-item pairs: it accepts any number of *arguments*, of any *size*. The following, for example, builds a list of three-item tuples for three arguments, with items from each sequence—essentially projecting by columns:

```
>>> T1, T2, T3 = (1, 2, 3), (4, 5, 6), (7, 8, 9)
>>> T3
(7, 8, 9)
>>> list(zip(T1, T2, T3))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

3 args of 3 vals => 3 3-item tuples

And formally speaking, for N arguments that contain M items, `zip` gives us an M -long series of N -ary tuples:

```
>>> list(zip(T1, T2))
[(1, 4), (2, 5), (3, 6)]
```

2 args of 3 vals => 3 2-item tuples

When argument lengths differ, `zip` *truncates* the series of result tuples at the length of the shortest sequence. To demo, the following zips two strings to pick out characters in parallel, but the result has only as many tuples as the length of the shortest sequence (formally again, M in the prior definition is really the minimum of arguments' lengths):

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> list(zip(S1, S2))                # Truncates at len(shortest)
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

To pad instead of truncating, you can write loop code to pad results yourself—as we will in [Chapter 20](#), after we’ve had a chance to study some additional iteration concepts that make it a fair fight.

More zip roles: dictionaries

Fine points aside, parallel traversals with `zip` are also useful in *dictionary* construction. We met this technique in [Chapter 8](#), but here’s a quick refresher in the context of looping statements. As we learned earlier, you can always create a dictionary by calling `dict`, assigning to keys over time, or coding a dictionary literal like the following:

```
>>> D1 = {'app': 1, 'script': 3, 'program': 5}          # Or dict(key=value,...)
>>> D1
{'app': 1, 'script': 3, 'program': 5}
```

What to do, though, if your program obtains dictionary keys and values at runtime, after you’ve coded your script? For example, the following may be collected from a user, a file, or any other dynamic source:

```
>>> keys = ['app', 'script', 'program']
>>> vals = [1, 3, 5]
```

One way to turn these into a dictionary is to `zip` the lists and step through them in parallel with a `for` loop:

```
>>> list(zip(keys, vals))
[('app', 1), ('script', 3), ('program', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v

>>> D2
{'app': 1, 'script': 3, 'program': 5}
```

As suggested earlier in this book, though, you can skip the `for` loop altogether in this context, and simply pass the zipped keys/values lists to the built-in `dict` constructor call:

```
>>> D3 = dict(zip(keys, vals))
>>> D3
{'app': 1, 'script': 3, 'program': 5}
```

The built-in name `dict` is really a *type* name (you’ll learn about type names, and subclassing them, in [Chapter 32](#)). Calls to it are object construction requests, but also perform a to-dictionary conversion here. In the next chapter, you’ll also learn more about related but richer concepts—list comprehensions, which build lists in expressions, and their dictionary comprehensions kin, which are an alternative to both `for` statements and `dict` for zipped key/value pairs:

```
>>> {k: v for (k, v) in zip(keys, vals)}
{'app': 1, 'script': 3, 'program': 5}
```

Offsets and Items: enumerate

Our final loop-helper function is designed to support dual usage modes. Earlier, we discussed using `range` to generate the offsets of items in a string, rather than the items at those offsets. In some programs, though, we need *both*: the item to use, plus an offset as we go. This might be coded with a `for` loop that also keeps a counter of the current offset:

```
>>> S = 'hack'
>>> offset = 0
>>> for item in S:
    print(item, 'appears at offset', offset)
    offset += 1

h appears at offset 0
a appears at offset 1
c appears at offset 2
k appears at offset 3
```

This works, but Python has a built-in function named `enumerate` that does the job for us—its net effect is to give loops a counter “for free,” without sacrificing the simplicity of automatic iteration:

```
>>> S = 'hack'
>>> for (offset, item) in enumerate(S):
    print(item, 'appears at offset', offset)

h appears at offset 0
a appears at offset 1
c appears at offset 2
k appears at offset 3
```

As for `range` and `zip`, the `enumerate` function’s result is an *iterable*—a kind of object that supports the iteration protocol that we will dive into in the next chapter. In short, it has a method called by the next built-in function, which returns an (*index*, *value*) tuple each time through the loop. The `for` steps through these tuples automatically, which allows us to unpack their values with tuple assignment, much as we did for `zip`:

```
>>> E = enumerate(S)
>>> E
<enumerate object at 0x10ebd7880>
>>> next(E)
(0, 'h')
>>> next(E)
(1, 'a')
>>> next(E)
(2, 'c')
```

We don’t normally see this machinery because all iteration contexts—including list comprehensions, the main subject of [Chapter 14](#)—run the iteration protocol automatically:

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'a', 'cc', 'kkk']

>>> for (ix, line) in enumerate(open('data.txt')):
    print(f'{ix} {line.rstrip()}')

0) Testing file IO
1) Learning Python, 6E
2) Python 3.12
```

To fully understand iteration concepts like `enumerate` and list comprehensions, though, we need to move on to the next chapter for a deeper dissection.

Chapter Summary

In this chapter, we explored Python's looping statements and their related tools. We looked at the `while` and `for` loop statements in depth, and we learned about their associated `else` clauses. We also studied the `break` and `continue` statements, which have meaning only inside loops, and met several built-ins commonly used in `for` loops, including `range`, `zip`, and `enumerate`, although some of the details regarding their roles as iterables were intentionally cut short.

In the next chapter, we continue the iteration story by discussing list comprehensions and the iteration protocol in Python—concepts strongly related to `for` loops. There, we'll also fill in the rest of the picture behind the iterable tools we met here, such as `range` and `zip`, and study some of the subtleties of their operation. As always, though, before moving on let's exercise the knowledge you've picked up here with a quiz.

Test Your Knowledge: Quiz

1. What are the main functional differences between `while` and `for` loops?
2. What's the difference between `break` and `continue`?
3. When is a loop's `else` clause executed?
4. How can you code a counter-based loop in Python?
5. What can a `range` be used for in a `for` loop?

Test Your Knowledge: Answers

1. The `while` loop is a general looping statement, but the `for` is designed to automatically iterate across items in a sequence or other iterable. Although the `while` can imitate the `for` with counter loops, it takes more code and might run slower.
2. The `break` statement exits a loop immediately (control flow winds up below the entire `while` or `for` loop statement), and `continue` jumps back to the top of the loop (control flow winds up positioned just before the test in `while` or the next item fetch in `for`).
3. The `else` clause in a `while` or `for` loop will be run once as the loop is exiting, if and only if the loop exits normally (i.e., by a false test in `while` or an empty object in `for`), without running into a `break` statement. A `break` exits the loop immediately, skipping the `else` part on the way out (if there is one).
4. Counter loops can be coded with a `while` statement that keeps track of the index manually, or with a `for` loop that uses the `range` built-in function to generate successive integer offsets. Neither is the preferred way to code in Python, if you need to simply step across all the items in a sequence. Instead, use a simple `for` loop without `range` or counters, whenever possible; it will be easier to code and usually quicker to run.

5. The `range` built-in can be used in a `for` loop to implement a fixed number of repetitions, to scan by offsets instead of items at offsets, to skip successive items as you go, and to change a list while stepping across it. None of these roles requires `range`, and most have alternatives—scanning actual items, three-limit slices, and list comprehensions are often better solutions today (despite the natural inclinations of ex-C programmers to want to count things!).

Why You Will Care: File Scanners

Loops come in handy anywhere you need to repeat an operation or process something more than once. Because *text files* contain multiple characters and lines, they are a typical role for loops. Assuming a file's contents can fit in memory, you can load it all at once with the file object's `read` method of [Chapter 9](#):

```
file = open('data.txt')           # Read contents into a string all at once
print(file.read())
```

For more granular access, you can scan by *characters* instead with either of the following—the second of which doesn't load the whole file and has grown terser with the `:=` named assignment of [Chapter 11](#):

```
for char in open('data.txt').read():
    print(char, end='')

file = open('data.txt')
while char := file.read(1):      # Read by character, empty means end-of-file
    print(char, end='')          # Don't add a \n after each character
```

To read by *lines* or *blocks* instead, you can use `while` loops like the following; binary data is often read by blocks using binary file mode `'rb'`, but text should use text mode to avoid splitting character bytes:

```
file = open('data.txt')
while line := file.readline():   # Read line by line
    print(line.rstrip())         # Line already has a \n newline

file = open('data.txt')
while chunk := file.read(10):    # Read block by block: up to 10 characters
    print(chunk, end='')         # Keep but don't add newlines
```

To read text files by *lines*, though, the `for` loop tends to be easiest to code and may be quickest to run:

```
for line in open('data.txt').readlines():
    print(line.rstrip())

for line in open('data.txt'):    # Use iterators: best for text input (maybe)
    print(line.rstrip())
```

The first version here uses the file `readlines` method to load a file all at once into a line-string list, but the second example relies on file *iterators* to automatically read one line on each loop iteration.

The second example is also generally best for text files—besides its simplicity, it works for arbitrarily large files because it doesn't load the entire file into memory all at once. The iterator version may also be the quickest, though speed can vary per Python release (we'll study ways to time code later in this book).

File `readlines` calls can still be useful, though—to *reverse* a file’s lines, for example, assuming its content can fit in memory. The `reversed` built-in works on sequences, but does not accept iterables that generate values; `sorted`, by contrast, does, so it can order all the lines in a file without loading it in full:

```
for line in reversed(open('data.txt').readlines()):
    print(line.rstrip())

for line in sorted(open('data.txt')):
    print(line.rstrip())
```

See Python’s documentation for more on the file-object calls used here—as well as its coverage of tools like `os.popen` that returns a file object connected to a *shell command*’s output (by default), and hence supports the same sort of loops. There’s also an `os.popen` example in [Chapter 21](#), and more on the distinctions of text and binary files in [Chapter 37](#) when we dive into Unicode more deeply.

Iterations and Comprehensions

In the prior chapter, we met Python’s two looping statements, `while` and `for`. Although they can handle most repetitive tasks programs need to perform, iterating over collections is so common and pervasive that Python provides additional tools to make it simpler and more efficient. This chapter begins our exploration of these tools. Specifically, it presents Python’s *iteration protocol*, a method-call model used by the `for` loop, and fills in some details on *comprehensions*, which are a close cousin to the `for` loop that apply an expression to each item in a collection.

Because these tools are related to both the `for` loop and functions, we’ll take a two-pass approach to covering them in this book—along with a postscript:

- *This chapter* introduces their basics in the context of looping-based tools, serving as something of a continuation of the prior chapter.
- **Chapter 20** revisits them in the context of function-based tools, and extends the topic to include built-in and user-defined *generators*.
- **Chapter 30** provides a shorter final installment in this story, which will show us how to code user-defined iterable objects with *classes*.

One note up front: some of the concepts presented in these chapters may seem advanced at first glance. With practice, though, you’ll find that these tools are useful and natural. Although never strictly required, they’ve also become commonplace in Python code, so a basic understanding can help if you must read programs written by others.

Iterations

In the preceding chapter, we learned that the `for` loop can work on any sequence type in Python, including lists, tuples, and strings, like this (with blank lines required by REPLs after compound statements omitted for brevity):

```
>>> for x in [1, 2, 3, 4]: print(x ** 2, end=' ')
1 4 9 16

>>> for x in (1, 2, 3, 4): print(x ** 3, end=' ')
1 8 27 64

>>> for x in 'text': print(x * 2, end=' ')
tt ee xx tt
```

As we’ve also learned, the `for` loop is even more generic than this—it works on any *iterable* object. In fact, this is true of all iteration *tools* that scan objects from left to right in Python, including `for` loops; comprehensions of all stripes; some `in` membership tests; the `zip`, `enumerate`, and `map` built-in functions; and more. Any iterable object will do, even nonsequences like dictionaries:

```
>>> for k in dict(a=1, b=2, c=3): print(k, end=' ')
a b c
```

The concept of iterable objects was added to Python after its inception, but it has come to permeate the language’s toolset. It’s essentially a generalization of the notion of sequences—an object is considered *iterable* if it is either a physically stored sequence or an object that produces one result at a time in the context of an iteration tool like `for`.

In a sense, iterable objects include both real sequences and *virtual* sequences computed on demand. The virtual sequences both save memory and avoid delays by producing results one at a time, instead of all at once. These are not true sequences, however: virtual iterables do not support the full range of operations defined for lists and tuples. Rather, they simply materialize a series of values over time and on request.

Whether an iterable is physical or virtual, it announces its support for iterations by implementing the *iteration protocol*—a set of callable methods used by all iteration tools, and the subject of the next section.



Terminology moment: The Python world sometimes uses the terms “iterable” and “iterator” interchangeably (and confusingly!) to refer to an object that supports iteration in general. For clarity, this book uses the term *iterable* to refer to an object that has the `iter` call at the top of the protocol we’re about to meet, and *iterator* to refer to an object that has the `next` call to produce results.

That is, an *iterable* returns an *iterator* that advances on `next`. This book also uses the phrase *iteration tool* for language tools that *run* an iteration, like `for` loops and `zip` calls. **Chapter 20** will muddle this jargon with the term *generator*—which refers to objects that automatically support the iteration protocol, and hence are iterable—even though all iterables generate results!

The Iteration Protocol

One of the easiest ways to understand the iteration protocol is to see how it works with a built-in type such as the *file* object we first explored in [Chapter 9](#). In this chapter, we'll be using the following three-line input file as a demo:

```
>>> print(open('data.txt').read())
Testing file IO
Learning Python, 6E
Python 3.12

>>> open('data.txt').read()
'Testing file IO\nLearning Python, 6E\nPython 3.12\n'
```

Recall from [Chapter 9](#) that open file objects have a method called `readline`, which reads one line of text from a file at a time—each time we call the `readline` method, we advance to the next line. At the end of the file, an empty string is returned, which we can detect to break out of a line-reading loop (remember, empty means false):

```
>>> f = open('data.txt')           # Read a three-line file in this directory
>>> f.readline()                   # readline loads one line on each call
'Testing file IO\n'
>>> f.readline()                   # Newlines are \n everywhere in text mode
'Learning Python, 6E\n'
>>> f.readline()                   # Last lines may have a \n or not
'Python 3.12\n'
>>> f.readline()                   # Returns empty string at end-of-file
''
```

Files, however, also have a method named `__next__` that has a nearly identical effect—it returns the next line from a file each time it is called. The only noticeable difference is that `__next__` raises a built-in `StopIteration` *exception* (that is, invokes a signaling event) at end-of-file instead of returning an empty string:

```
>>> f = open('data.txt')           # __next__ loads one line on each call too
>>> f.__next__()                   # But raises an exception at end-of-file
'Testing file IO\n'
>>> f.__next__()
'Learning Python, 6E\n'
>>> f.__next__()
'Python 3.12\n'
>>> f.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

And this interface is most of what we call the *iteration protocol* in Python. Any object with a `__next__` method to advance to a next result, which raises `StopIteration` at the end of the series of results, is considered an *iterator* in Python. Any such object may also be stepped through with a `for` loop or any other iteration tool, because all iteration tools normally work internally by calling `__next__` on each iteration and catching the `StopIteration` exception to know when to exit. As you'll see in a moment, for some objects the full protocol includes an additional first step to call `iter`, but this isn't required for files.

The upshot of all this magic is that, as mentioned in Chapters 9 and 13, the generally best way to read a text file line by line today is to *not read it at all*—instead, allow the for loop to automatically call `__next__` to advance to the next line on each iteration. The file object’s iterator will do the work of automatically loading lines as you go, both one at a time and efficiently. The following, for example, reads line by line, printing the uppercase version of each line along the way—without ever explicitly reading from the file at all:

```
>>> for line in open('data.txt'):          # Use file iterators to read by lines
    print(line.upper(), end='')            # Calls __next__, catches StopIteration

TESTING FILE IO
LEARNING PYTHON, 6E
PYTHON 3.12
```

Notice that the print uses `end=''` here to suppress adding a `\n`, because line strings already have one (without this, our output would be double-spaced). This for coding pattern is usually the best way to read text files line by line, for three reasons: it’s the simplest to code, might be the quickest to run, and uses memory space sparingly. Prior to the advent of the iteration protocol in Python, programmers achieved the same effect with a for loop by calling the file `readlines` method to load the file’s content into memory as a *list* of line strings:

```
>>> for line in open('data.txt').readlines():
    print(line.upper(), end='')

TESTING FILE IO
LEARNING PYTHON, 6E
PYTHON 3.12
```

This `readlines` technique still works, but is not considered the best practice today because it performs poorly in terms of memory usage. In fact, because this version really does load the entire file into memory all at once, it will not even work for files too big to fit into the memory space available on your device. By contrast, the iterator-based version is immune to such memory-explosion issues because it reads just one *line* at a time. The iterator version might run quicker too, though this can vary per Python release (but see the upcoming note for a few specs).

As mentioned in the prior chapter’s closing sidebar, “[Why You Will Care: File Scanners](#)” on page 333, it’s also possible to read a file line by line with a `while` loop:

```
>>> f = open('data.txt')
>>> while line := f.readline():
    print(line.upper(), end='')

TESTING FILE IO
LEARNING PYTHON, 6E
PYTHON 3.12
```

However, this may run slower than the iterator-based for loop version because file iterators run at C-language speed inside the standard CPython, whereas the `while` version must run Python bytecode through the Python virtual machine. Anytime we trade Python code for C code, speed tends to increase. This is not an absolute truth, though; again, we’ll explore timing techniques in [Chapter 21](#) for measuring the relative speed of alternatives like these, though the following note ruins some of the surprise for the impatient.



Spoiler alert: Per calls to `min(timeit.repeat(code, repeat=50, number=10))` in CPython 3.12 on macOS, the file iterator is still slightly faster than `readlines`, which is faster than the `while` loop. With a 9k-line file and this chapter's code (using `pass` for loop bodies), the iterator, `readlines`, and `while` alternatives check in at 0.0073, 0.0077, and 0.0102 seconds, respectively. The `while` is slowest and using `:=` doesn't help much (it's 0.0104 sans `:=`). For more info, see the examples package and [Chapter 21](#). Caveats: your test variables may vary, memory matters too, and 0.0029 seconds may not be enough to get excited about in some programs.

The iter and next built-ins

To simplify manual iteration code, Python also provides a built-in function, `next`, that has the same net effect as calling an object's `__next__` method. That is, given an iterator object `X`, the call `next(X)` is the same as `X.__next__()`, but is noticeably simpler to type and read (and actually runs slightly faster in CPython 3.12 for tested cases). With files, for instance, either form may be used:

```
>>> f = open('data.txt')
>>> f.__next__()                # Call iteration method directly
'Testing file IO\n'
>>> next(f)                    # next(f) is the same as f.__next__()
'Learning Python, 6E\n'
>>> next(f)
'Python 3.12\n'
>>> next(f)
...exception text omitted from here on...
StopIteration
```

Technically, there is one more piece to the iteration protocol alluded to earlier. When the `for` loop begins, it first obtains an *iterator* from an *iterable* object, by calling the iterable's `__iter__` method. The object returned by this call in turn has the required `__next__` method to advance. For convenience again, the `iter` built-in function internally runs the equivalent of the `__iter__` method, much as `next` runs the equivalent of `__next__`.

Hence, `for` loops run the internal equivalent of the following, though the `iter` step is moot and optional for files—they are their own iterators, because files don't support multiple scans per open:

```
>>> f = open('data.txt')
>>> I = iter(f)                # Fetch an iterator from an iterable
>>> next(I)                   # Fetch the next result from the iterator
'Testing file IO\n'
>>> next(I)                   # Files iterables are iterators themselves
'Learning Python, 6E\n'
...etc...
```

The full iteration protocol

With all these pieces in place, [Figure 14-1](#) sketches this full iteration protocol, used by every iteration tool in Python and supported by a wide variety of object types. It's based on *two objects* used in two distinct steps by iteration tools:

- The *iterable* object for which iteration is requested. Calling this object's `__iter__` returns an iterator, and is the same as calling `iter`.

- The *iterator* object returned by the iterable. Calling this object's `__next__` produces results during the iteration and raises `StopIteration` when no more results remain, and is the same as calling `next`.

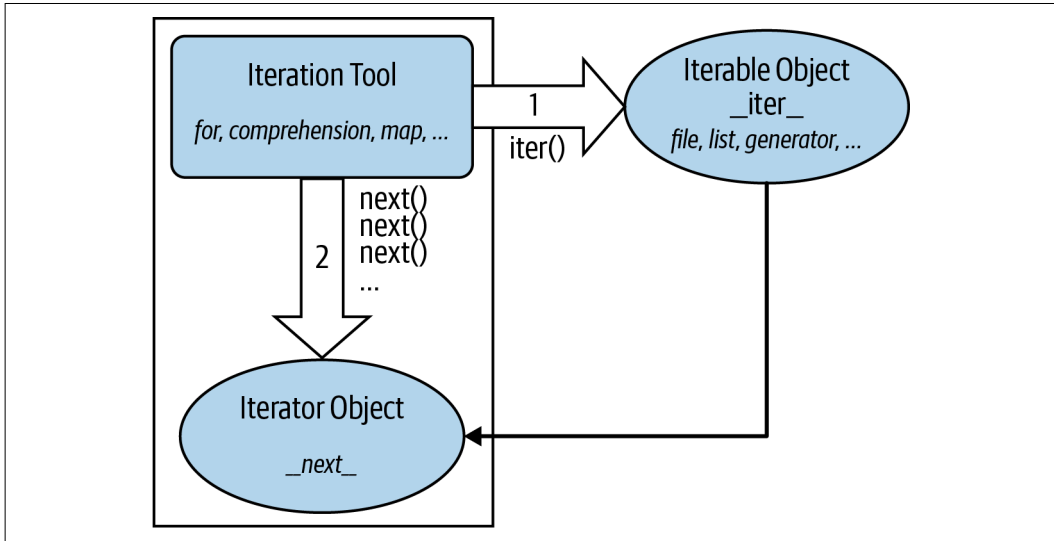


Figure 14-1. The iteration protocol, used by *for* loops, comprehensions, maps, and more

These steps are orchestrated automatically by iteration tools in most cases, but it helps to understand these two objects' roles. For example, in some cases these two objects are the *same* when only a single scan is supported (e.g., files), and the *iterator* object is often a temporary, used internally by the iteration tool.

Moreover, some objects are *both* an iteration *tool* (they iterate) and an iterable *object* (their results are iterable)—including the `enumerate` and `zip` built-ins, and [Chapter 20](#)'s generator expressions. Such iterables already avoid constructing result lists in memory themselves, but applying them to other iterables saves even more space. When such tools are combined, no work is done until an iteration tool requests results.

In code, the protocol's first step becomes obvious if we look at how *for* loops internally process built-in sequence types such as lists (*for* uses the internal equivalents of the “`__`” methods, but you use either in your code):

```

>>> L = [1, 2]
>>> I = iter(L)                # Obtain an iterator object from an iterable
>>> next(I)                    # Call next(iterator) to advance to next item
1
>>> next(I)                    # Or use L.__iter__() and I.__next__() calls
2
>>> next(I)
StopIteration

```

As we saw earlier, the initial `iter` step is not required for files, because a file object supports just one scan and hence is its own iterator. You can see this yourself with `is` (recall from [Chapter 9](#) that this means object *identity*—the same exact piece of object memory, not just the same value):

```
>>> f = open('data.txt')
>>> iter(f) is f           # Files are iterators themselves: iter() optional
True
>>> iter(f) is f.__iter__() # Both calls return file object f itself
True
>>> next(f)                # Which responds to next requests directly
'Testing file IO\n'
```

Lists and many other built-in objects, though, are not their own iterators because they do support multiple open iterations—there may be any number of iterations active in nested loops, and all may be at different positions at a given point in time. For such objects, we must call `iter` to start iterating manually:

```
>>> L = [1, 2, 3]
>>> iter(L) is L           # Lists are not their own iterators: use iter()
False
>>> next(L)
TypeError: 'list' object is not an iterator

>>> I = iter(L)             # Same as L.__iter__()
>>> next(I)                 # Same as I.__next__()
1
```

Manual iteration

Although Python iteration tools call these functions automatically, we can also use them to apply the iteration protocol manually when needed. The following demonstrates the equivalence between automatic and manual iteration (again, for runs the internal equivalent of `I.__next__` instead of the `next(I)` used here, but the effect is the same):

```
>>> L = [1, 2, 3]
>>>
>>> for X in L:              # Automatic iteration
    print(X ** 2, end=' ')   # Obtain iter, call __next__, catch exceptions

1 4 9

>>> I = iter(L)              # Manual iteration: what for loops usually do
>>> while True:              # Use try statements to catch stop exception
    try:
        X = next(I)
    except StopIteration:
        break
    print(X ** 2, end=' ')

1 4 9
```

To understand this code, you need to know that `try` statements run an action and catch exceptions that occur while the action runs (we met exceptions briefly in [Chapter 10](#) but will explore them in depth in [Part VII](#)).

More on `iter` and `next`

For full fidelity, it should also be noted that `for` loops and other iteration tools can sometimes work differently for user-defined classes—repeatedly *indexing* an object instead of running the iteration protocol—but they prefer the iteration protocol when supported (more on this story when we study operator overloading in [Chapter 30](#)).

Though not commonly used, it's also worth noting that `next` accepts an optional second *default* argument for an exit value; if passed, it's returned at the end instead of raising a stop exception:

```
>>> L = [1]
>>> I = iter(L)                                # Result instead of exception
>>> next(I, 'end of list')
1
>>> next(I, 'end of list')
'end of list'
```

Combined with the `:=` named-assignment expression, this can shave multiple lines off the preceding manual-iteration code—but will also fail if the passed default can appear as a valid result:

```
>>> I = iter(L)
>>> while (X := next(I, None)) != None:        # Same effect, less code
    print(X ** 2, end=' ')                    # Assuming None is safe!
```

Though also uncommon, `iter` accepts a second *sentinel* argument to signal stop from a callable. This very different mode provides an arguably tricky way to use `for` to read files by blocks—but requires info on functions or `lambda`, which we don't yet have:

```
>>> f = open('data.txt')
>>> I = iter(lambda: f.read(5), '')            # Callables and sentinels
>>> for block in I: print(block, end='')        # Assuming you know lambda!
```

Watch for `lambda` in the next part of this book, and see Python's docs for more on `next` and `iter` modes.

Other Built-in Iterables

Besides files and physical sequences like lists, many other objects in Python have useful iterators as well. Now that we have a better handle on how the iteration protocol works, this section revisits some tools we've already seen in this context, and introduces a handful of additional iterables along the way.

Reprise: Dictionaries, `range`, `enumerate`, and `zip`

As we saw in the last chapter, the usual way to step through the keys of a dictionary is with a `for` loop:

```
>>> D = dict(a=1, b=2)
>>> for key in D:                               # Dictionaries are implicitly iterable
    print(key, D[key])

a 1
b 2
```

This works simply because dictionaries are iterables with an iterator that automatically returns one key at a time in an iteration tool like `for`:

```
>>> I = iter(D)                                # Which just means they support the protocol
>>> next(I)
'a'
>>> next(I)
'b'
>>> next(I)
StopIteration
```

The iteration protocol is also the reason that we've had to wrap `range` results in a `list` call to see their values all at once in a REPL. Objects that are nonsequence iterables return results one at a time, not in a physical list:


```

>>> R = range(5)
>>> R
range(0, 5)
>>> I = iter(R)                # Use iteration protocol to produce results
>>> next(I)
0
>>> next(I)
1
>>> list(range(5))            # Or use list() to collect all results at once
[0, 1, 2, 3, 4]

```

Note that the `list` call here is not needed *and shouldn't be used* in contexts where iteration happens automatically—such as within `for` loops. It is needed for displaying values all at once, though, and may also be required when list-like behavior or multiple scans are required for objects that normally produce results on demand (more on this later).

Now that you have a better understanding of this protocol, you should also be able to see how it explains why the `enumerate` tool introduced in the prior chapter works the way it does:

```

>>> E = enumerate('text')    # enumerate is an iterable too
>>> E
<enumerate object at 0x1010ab880>
>>> I = iter(E)
>>> next(I)                  # Generate results with iteration protocol
(0, 't')
>>> next(I)                  # Or use list() to force generation to run
(1, 'e')
>>> list(enumerate('text'))
[(0, 't'), (1, 'e'), (2, 'x'), (3, 't')]

```

Unlike `range`, the `enumerate` built-in's result is its own iterator, much like files. This means it supports just one scan per call, and `iter` is optional (more on this ahead too):

```

>>> R = range(5)
>>> iter(R) is R
False

>>> E = enumerate('text')
>>> iter(E) is E
True
>>> next(E)
(0, 't')

```

The `zip` built-in, covered in the prior chapter, works the same way in iteration tools. Like `enumerate`, `zip` is also its own iterator, so we have to call it again to iterate again:

```

>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> Z
<zip object at 0x101236240>
>>> I = iter(Z)
>>> next(I)
(1, 10)
>>> next(I)
(2, 20)

>>> I is Z
True
>>> list(Z)
[(3, 30)]
>>> list(zip((1, 2, 3), (10, 20, 30)))
[(1, 10), (2, 20), (3, 30)]

```

Iterator nesting

More interestingly, `zip` is *both* iteration tool and iterable: it iterates over its arguments' results, but also returns an iterable object with an iterator for its own results. In the following, for example, it's a tool that *receives* results from two `range` iterables, but its own results are *produced* on demand as well. In other words, there are three iterables and *two levels* of iteration at work here:

```
>>> Z = zip(range(1, 4), range(10, 40))
>>> next(Z)
(1, 10)
>>> next(Z)
(2, 11)
>>> next(Z)
(3, 12)
>>> list(zip(range(1, 4), range(10, 40)))
[(1, 10), (2, 11), (3, 12)]
```

In fact, iterators can be nested arbitrarily. Because `enumerate` is both iteration tool and object too, in the following, `range`, `enumerate`, and `zip` all produce their results on demand, and `list` makes all the dances run:

```
>>> list(enumerate(range(1, 4)))
[(0, 1), (1, 2), (2, 3)]

>>> list(zip(enumerate(range(1, 4)), enumerate(range(5, 8))))
[((0, 1), (0, 5)), ((1, 2), (1, 6)), ((2, 3), (2, 7))]
```

Similarly, the following enumerates the zipped results of ranges—but only when requested by `for`:

```
>>> for x in enumerate(zip(range(1, 4), range(5, 8))): print(x)
...
(0, (1, 5))
(1, (2, 6))
(2, (3, 7))
```

There are *three* levels of iterables in this, all deferring their results until they are activated. In practice, this works naturally, but nothing happens in such code until a tool like `list` or `for` asks for results at the top. In response, all the actors here return just one result at a time, to minimize memory requirements and avoid delays.

Functional iterables: map and filter

Like `range`, `enumerate`, and `zip`, the `map` and `filter` built-ins produce their results individually to conserve space and avoid pauses. Like `enumerate` and `zip`, these tools also are both iterable tools and iterable objects themselves: they scan other iterables, and produce their own results on demand.

Unlike other iterables we've met, though, `map` and `filter` apply *function calls* instead of expressions, so their complete story requires function-coding skills we won't gain until the next part of the book. Still, we can preview their fundamentals here using built-in functions without having to code new functions of our own.

For example, the `map` built-in, which made a brief cameo appearance in [Chapter 8](#) (and has nothing directly to do with mappings like dictionaries!), calls a provided function for each item in a provided iterable, and returns the collected results as another iterable. In the following, it applies the `ord` built-in to collect character code points:

```

>>> ord('p')                # Return a single character's code point
112
>>> M = map(ord, 'py3')      # map returns an iterable, not a list
>>> M                        # Runs ord(x) for every x in iterable
<map object at 0x101227550>
>>> next(M)                  # Iterating manually: exhausts results
112                          # map supports no sequence ops like [i]
>>> next(M)
121
>>> next(M)
51
>>> next(M)
StopIteration

```

As usual, you can force results with `list` if you must treat them as a list, and for automates iterations:

```

>>> list(map(ord, 'py3'))    # Force a real list - only if needed
[112, 121, 51]

>>> M = map(ord, 'py3')      # Must call again to scan again
>>> for x in M: print(x, end=' ') # Iteration tools auto call next()
112 121 51

```

The filter built-in, which we met momentarily in [Chapter 12](#) and will study more fully in the next part of this book, is analogous. It returns items in an iterable for which a passed-in function returns `True`. In the following, we’re leveraging concepts we’ve already learned—`True` includes nonempty and nonzero objects, the `bool` built-in returns a single object’s truth value, and the `str` string’s `isdigit` method is true for all-digit text:

```

>>> filter(bool, ['lp6e', '', 2024])
<filter object at 0x101227850>

>>> list(filter(bool, ['lp6e', '', 2024]))    # Collect "true" items
['lp6e', 2024]

>>> list(filter(str.isdigit, ['lp6e', '2024'])) # Collect all-digit strings
['2024']

```

Like most of the tools discussed in this section, `filter` both *accepts* an iterable to process and *returns* an iterable to generate results. It doesn’t do any work until code like a `for` loop asks it to. As a preview, both `map` and `filter` can be emulated roughly with list *comprehensions* and more closely with [Chapter 20](#)’s *generator* expressions; but to grok the following code in full, we have to await this chapter’s presentation of comprehensions coming up soon:

```

>>> [ord(x) for x in 'py3']
[112, 121, 51]

>>> [x for x in ['lp6e', '2024'] if x.isdigit()]
['2024']

```

Multiple-pass versus single-pass iterables

We’ve noted a few times that some iterables that don’t allow multiple scans are their own iterables. Since this is a subtle difference that can impact the way you’ll use them, it’s worth a separate callout here.

In particular, the `range` built-in’s result, along with objects like dictionaries and lists, differs from other built-ins described in this section. They are *not* their own iterators (you must make one with `iter` when iterating manually), and they support multiple iterators (each remembers its position independently):

```

>>> R = range(3)                                # range allows multiple iterators
>>> next(R)
TypeError: range object is not an iterator

>>> I1 = iter(R)
>>> next(I1)
0
>>> next(I1)
1
>>> I2 = iter(R)                                # Two iterators on one range
>>> next(I2)
0
>>> next(I1)                                    # I1 is at a different spot than I2
2

```

By contrast, the results of `enumerate`, `zip`, `map`, `filter`, as well as `open` for files, *are* their own iterators, because none of these tools support multiple active iterations on the same call result. Because of this, the `iter` call is optional for stepping through such objects' results (though harmless: their `iter` is themselves), and we must call these tools again to begin a fresh iteration from the start:

```

>>> Z = zip((1, 2, 3), (10, 11, 12))
>>> I1 = iter(Z)
>>> I2 = iter(Z)                                # Two iterators on one zip
>>> next(I1)
(1, 10)
>>> next(I1)
(2, 11)
>>> next(I2)                                    # But I2 is at same spot as I1!
(3, 12)

>>> M = map(ord, 'py3')                         # Ditto for map (and others)
>>> I1, I2 = iter(M), iter(M)
>>> next(I1), next(I1)
(112, 121)
>>> next(I2)
51

>>> L = [0, 1, 2]                                # But lists (and others) do many scans
>>> I1, I2 = iter(L), iter(L)
>>> next(I1), next(I1)
(0, 1)
>>> next(I2)                                    # Multiple active scans, like range
0

```

When we code our own iterable objects with classes later in the book ([Chapter 30](#)), you'll see that multiple iterators are usually supported by returning *new* objects for the `iter` call; a single iterator generally means an object returns *itself* and supports `next` directly. In [Chapter 20](#), you'll also find that *generator* functions and expressions behave like `map` and `zip` instead of `range` in this regard, supporting just a single active iteration scan. Also in that chapter, you'll see some subtle implications of single-pass iterators in loops that attempt to scan multiple times—code that treats these as lists may fail without manual list conversions.

Standard-library iterables in Python

Finally, while out of scope here, and technically part of its standard library instead of its language, Python provides additional tools that support the iteration protocol and thus may also be used in for loops and other iteration tools.

For instance, *shelves* (an access-by-key filesystem for Python objects), as well as the results of `os.popen` (a tool for reading the output of shell commands), are iterables that can be processed with the full set of iteration tools. The standard directory (a.k.a. folder) walker in Python, `os.walk`, is iterable too, but we'll save details and an example until [Chapter 20](#)'s coverage of this tool's basis—generators and `yield`.

Ultimately, all such tools implement the `iter/next` interface defined by the iteration protocol. We don't normally see this machinery because `for` and its kin run it for us automatically to step through results. In fact, everything that scans left to right in Python employs the iteration protocol in the same way—including the topic of the next section.

Comprehensions

Now that we've seen how the iteration protocol works, let's turn to one of its most common use cases. Together with `for` loops, list *comprehensions* are one of the most prominent contexts in which the iteration protocol is applied.

In the previous chapter, we learned how to use `range` to change a list as we step across it:

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in range(len(L)):
    L[i] += 10

>>> L
[11, 12, 13, 14, 15]
```

This works, but as mentioned there, it may not be the optimal “best practice” approach in Python. Today, the list comprehension expression makes many such prior coding patterns obsolete. Here, for example, we can replace the loop with a single expression that produces the desired result list:

```
>>> L = [x + 10 for x in L]
>>> L
[21, 22, 23, 24, 25]
```

The net result is similar, but it requires less coding on our part and is likely to run substantially faster—in fact, it's often *twice* as fast as tested in CPython 3.12. The list comprehension isn't exactly the same as the `for` loop statement version because it makes a *new* list object, which might matter if there are multiple references to the original list, but it's close enough for most applications and is a common and convenient enough approach to merit a closer look here.

List Comprehension Basics

The list comprehension was introduced in [Chapter 4](#), and it's been demoed often. Syntactically, its syntax is derived from a construct in set theory notation that applies an operation to each item in a set, but you don't have to know set theory to use this tool. In Python, most people find that a list comprehension simply looks like a backward `for` loop.

To get a handle on the syntax, let's dissect the prior section's example in more detail:

```
L = [x + 10 for x in L]
```

List comprehensions are written in square brackets because they are ultimately a way to construct a new list. They begin with an arbitrary expression that we make up, which uses a loop variable that we make up (`x + 10`). That is followed by what you should now recognize as the header of a `for` loop, which names the loop variable, and an iterable object (`for x in L`).

To run the expression, Python executes an iteration across `L` inside the interpreter, assigning `x` to each item in turn, and collects the results of running the items through the expression on the left side. The result list we get back is exactly what the list comprehension says—a new list containing `x + 10`, for every `x` in `L`.

Technically speaking, list comprehensions are never really required because we can always build up a list of expression results manually with `for` loops that append results as we go:

```
>>> res = []
>>> for x in L:
    res.append(x + 10)

>>> res
[31, 32, 33, 34, 35]
```

In fact, this is exactly what the list comprehension does internally (using internal equivalents, of course).

However, list comprehensions are more concise to write, and widely useful in Python programs because building result lists is such a common task. Moreover, depending on your Python and code, list comprehensions might run much faster than manual `for` loop statements (often 2X as stated earlier) because their iterations are performed at the speed of optimized (and usually compiled) code inside the interpreter, rather than with manual Python code. Especially for larger data sets, there is often a major performance advantage to using this expression.

List Comprehensions and Files

Let's work through another common application of list comprehensions to explore them in more detail. Recall that the file object has a `readlines` method that loads the file into a list of line strings all at once:

```
>>> f = open('data.txt')
>>> lines = f.readlines()
>>> lines
['Testing file IO\n', 'Learning Python, 6E\n', 'Python 3.12\n']
```

This works as we saw earlier, but the lines in the result all include the newline character (`\n`) at the end. For many programs, the newline character gets in the way—we have to be careful to avoid double-spacing when printing, and so on. It would be nice if we could get rid of these newlines all at once, wouldn't it?

Anytime we start thinking about performing an operation on each item in a sequence, we're in the realm of list comprehensions. For example, assuming the variable `lines` is as it was in the prior interaction, the following code does the job by running each line in the list through the string `rstrip` method to remove whitespace on the right side (a `line[:-1]` slice would work, too, but only if we can be sure all lines are properly `\n` terminated, and this may not always be the case for the last line in a file):

```
>>> lines = [line.rstrip() for line in lines]
>>> lines
['Testing file IO', 'Learning Python, 6E', 'Python 3.12']
```

This works as planned. Because list comprehensions are an iteration *tool* just like `for` loop statements, though, we don't even have to open the file ahead of time. If we open it inside the expression, the list comprehension will automatically use the iteration protocol we met earlier in this chapter. That is, it will read one line from the file at a time by calling the file's next handler method, run the line through the `rstrip` expression, and add it to the result list. Again, we get what we ask for—the `rstrip` result of a line, for every line in the file:

```
>>> lines = [line.rstrip() for line in open('data.txt')]
>>> lines
['Testing file IO', 'Learning Python, 6E', 'Python 3.12']
```

This expression does a lot implicitly, but we’re getting a lot of work for free here—Python scans the file line by line and builds a list of operation results automatically. It’s also an *efficient* way to code this operation: because most of this work is done inside the Python interpreter, it’s likely faster than an equivalent `for` statement. Just as importantly, its use of file iterators means that it won’t load the file into memory all at once, like `readlines` does. Again, especially for large files, the advantages of list comprehensions can be significant.

Besides their efficiency, list comprehensions are also remarkably expressive. In our example, we can run any string operation on a file’s lines as we iterate. To illustrate, here’s the list comprehension equivalent to the file iterator uppercase example we met earlier, along with a few other representative operations to sample the possibilities:

```
>>> [line.upper() for line in open('data.txt')]
['TESTING FILE IO\n', 'LEARNING PYTHON, 6E\n', 'PYTHON 3.12\n']

>>> [line.rstrip().upper() for line in open('data.txt')]
['TESTING FILE IO', 'LEARNING PYTHON, 6E', 'PYTHON 3.12']

>>> [line.split() for line in open('data.txt')]
[['Testing', 'file', 'IO'], ['Learning', 'Python,', '6E'], ['Python', '3.12']]

>>> [line.replace('\n', '!') for line in open('data.txt')]
['Testing file IO!', 'Learning Python, 6E!', 'Python 3.12!']

>>> [(('Py' in line, line.split()[0]) for line in open('data.txt'))]
[(False, 'Testing'), (True, 'Learning'), (True, 'Python')]
```

Recall that the method *chaining* in the second of these examples works because string methods return a new string, to which we can apply another string method. The last of these shows how we can also collect *multiple* results, as long as they’re wrapped in a collection like a tuple or list.

One fine point here: recall from [Chapter 9](#) that file objects *close* themselves automatically in CPython when garbage-collected if still open. Hence, these list comprehensions will also automatically close the file when their temporary file object is garbage-collected after the expression runs. Outside CPython, though, you may want to code these to close manually if this is run in a loop, to ensure that file resources are freed immediately: open before the comprehension, and close after. See [Chapter 9](#) for more on file close calls if you need a refresher on this.

Extended List Comprehension Syntax

Handy as they may already seem, list comprehensions can be even richer in practice, and even constitute a sort of *iteration mini-language* in their fullest forms. Let’s take a quick look at their extra syntax tools here.

Filter clauses: `if`

As one particularly useful extension, the `for` loop nested in a comprehension expression can have an associated `if` clause to *filter out* of the result items for which the test is not true. (It’s really a *filter in*, but it works out the same.)

For example, suppose we want to repeat the prior section's file-scanning example, but we need to collect only lines that begin with the letters *L* or *P* (perhaps the first character on each line is an action code of some sort). Adding a simple `if` filter clause to our expression does the trick:

```
>>> lines = [line.rstrip() for line in open('data.txt') if line[0] in 'LP']
>>> lines
['Learning Python, 6E', 'Python 3.12']
```

Here, the `if` clause checks each line read from the file to see whether its first character matches; if not, the line is omitted from the result list, and the iteration continues. This is a fairly big expression, but it's easy to understand if we translate it to its simple `for` loop statement equivalent:

```
>>> res = []
>>> for line in open('data.txt'):
    if line[0] in 'LP':
        res.append(line.rstrip())

>>> res
['Learning Python, 6E', 'Python 3.12']
```

In general, we can always translate a list comprehension to a `for` statement by *appending* as we go and further *indenting* each successive part. The converse isn't true: `for` statements are more general and can address additional roles out of scope for comprehensions, but the latter's results collection is a very common task.

This `for` statement equivalent works, but it takes up four lines instead of one and may run slower. In fact, you can squeeze a substantial amount of logic into a list comprehension when you need to—the following works like the prior but selects only lines that *end in a digit* (before the newline at the end), by filtering with a more sophisticated expression on the right side (which uses `[-1:]` instead of `[-1]` to handle files with blank lines empty after `rstrip()`):

```
>>> [line.rstrip() for line in open('data.txt') if line.rstrip()[-1:].isdigit()]
['Python 3.12']
```

As another `if` filter example, the first result in the following gives the total lines in a text file, and the second strips whitespace on both ends to *omit blank lines* in the tally in just one line of code:

```
>>> fname = 'data-blank-lines.txt'
>>> len(open(fname).readlines())                                # All lines
5
>>> len([line for line in open(fname) if line.strip() != ''])    # Nonblank lines
3
```

Nested loops: `for`

List comprehensions can become even more complex if we need them to—for instance, they may contain *nested loops*, coded as a series of `for` clauses. In fact, their full syntax allows for any number of `for` clauses, each of which can have an optional associated `if` clause.

For example, the following builds a list of the concatenation of `x + y` for every `x` in one string and every `y` in another. It effectively collects all the *ordered combinations* of the characters in two strings:

```
>>> [x + y for x in 'abc' for y in '123']
['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3']
```

Again, one way to understand this expression is to convert it to statement form by indenting its parts. The following is an equivalent, but likely slower, alternative way to achieve the same effect (it's the same

as the first nested for example in the prior chapter, but builds a list of results instead of simply printing them):

```
>>> res = []
>>> for x in 'abc':
    for y in '123':
        res.append(x + y)

>>> res
['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3']
```

Beyond this complexity level, though, list comprehension expressions can sometimes become too compact for their own good. In general, they are intended for simple types of iterations; for more involved work, a simpler for statement structure will probably be easier to understand and modify in the future. As usual in programming, if something is difficult for you to understand, it's probably not the best idea.

Comprehensions Cliff-Hanger

Because comprehensions are generally better taken in multiple doses, we'll cut this story short here for now. We'll revisit list comprehensions in [Chapter 20](#) in the context of functional programming tools, and will define their syntax more formally and explore additional examples there. As you'll find, comprehensions turn out to be just as related to *functions* as they are to looping *statements*.

List comprehensions are also related to—and predate—the *set* and *dictionary* comprehensions introduced in this book's prior part, as well as the *generator* expression you'll meet later that produces items on request instead of building a list. All share the same syntax, but are coded slightly differently and produce different sorts of stuff:

```
[x + 10 for x in L if x > 0]      # List comprehension
{x + 10 for x in L if x > 0}      # Set comprehension
{x: x + 10 for x in L if x > 0}   # Dictionary comprehension
(x + 10 for x in L if x > 0)      # Generator expression
```

We'll put the last three of these to work on files briefly in the next section. To better expand this plotline to generators, though, we have to move on to this book's next part.



Speed disclaimer: As a blanket qualification for all performance claims in this book, the relative speed of code depends much on the exact code tested and version of Python used, and is prone to vary and change. For example, list comprehensions have been consistently twice as fast as corresponding for loops on most tests for all CPython versions through 3.12, but may be just marginally quicker on some tests, and perhaps even slower when *if* filter clauses are used. You'll learn how to time your own code and Python in [Chapter 21](#). For now, keep in mind that absolutes in performance benchmarks are as elusive as consensus in open source projects.

Iteration Tools

Later in this book, you'll learn how user-defined classes can implement the iteration protocol too. Because of this, it's sometimes important to know which built-in tools make use of it—any tool that employs the iteration protocol will automatically work on any built-in type or user-defined class that provides it. This section closes out this chapter with a summary and sort of “grand finale” of tools in this domain.

So far, we've mostly seen iterators at work in the context of the `for` loop statement, because this part of the book is focused on statements. Keep in mind, though, that *every* built-in tool that scans from left to right across collection objects uses the iteration protocol. This includes the `for` loops we've seen:

```
>>> for line in open('data.txt'):
    print(line.upper(), end='')
```

```
TESTING FILE IO
LEARNING PYTHON, 6E
PYTHON 3.12
```

But also much more. For instance, the prior section's list *comprehensions* and the `map` built-in function we met earlier use the same protocol as their `for` loop cousin. When applied to a file, they both leverage the file object's iterator automatically to scan line by line, fetching an iterator with `__iter__` and calling `__next__` each time through:

```
>>> uppers = [line.upper() for line in open('data.txt')]
>>> uppers
['TESTING FILE IO\n', 'LEARNING PYTHON, 6E\n', 'PYTHON 3.12\n']

>>> list(map(str.upper, open('data.txt')))
['TESTING FILE IO\n', 'LEARNING PYTHON, 6E\n', 'PYTHON 3.12\n']
```

As we saw earlier, the `map` built-in applies a function call to each item in an iterable object. `map` is similar to a list comprehension but is more limited because it requires a function instead of an arbitrary expression. It also *returns* an iterable object, so we must wrap it in a `list` call to force it to give us all its values at once. Because `map`, like the list comprehension, is related to both `for` loops and functions, watch for its revival in [Chapter 20](#).

Many of Python's other built-ins process iterables, too. We've seen how `zip` combines items from iterables, `enumerate` pairs items in an iterable with relative positions, and `filter` selects items for which a function is true. In addition, `sorted` sorts items in an iterable, and `reduce` (now oddly relegated to a module) runs pairs of items in an iterable through a function. All of these *accept* iterables, and `zip`, `enumerate`, and `filter` also *return* an iterable like `map`. Here they are in action running the file's iterator automatically to read line by line:

```
>>> sorted(open('data.txt'))
['Learning Python, 6E\n', 'Python 3.12\n', 'Testing file IO\n']

>>> list(zip(range(99), open('data.txt')))
[(0, 'Testing file IO\n'), (1, 'Learning Python, 6E\n'), (2, 'Python 3.12\n')]

>>> list(enumerate(open('data.txt')))
[(0, 'Testing file IO\n'), (1, 'Learning Python, 6E\n'), (2, 'Python 3.12\n')]

>>> list(filter(bool, open('data.txt')))
['Testing file IO\n', 'Learning Python, 6E\n', 'Python 3.12\n']

>>> import functools, operator
>>> functools.reduce(operator.add, open('data.txt'))
'Testing file IO\nLearning Python, 6E\nPython 3.12\n'
```

All of these are iteration tools, but they have unique roles. We met `zip` and `enumerate` in this chapter; `filter` and `reduce` are in [Chapter 19](#)'s functional programming domain, so we'll defer their details for now; the point to notice here is their use of the iteration protocol for files and other iterables.

We first saw the `sorted` function used here at work in [Chapter 4](#), and we used it in [Chapter 8](#). `sorted` is a built-in that employs the iteration protocol—it’s like the original list `sort` method, but it returns the new sorted list as a result and runs on any iterable object. Notice that, unlike `map` and others, `sorted` returns an actual *list* instead of an iterable. Per the prior chapter’s closer, its `reversed` cohort returns an iterable but does not run the iteration protocol.

In general, though, *everything* in Python’s built-in toolset that scans object is defined to use the iteration protocol on their subject. This even includes tools such as the `list` and `tuple` built-in functions (which build new objects from iterables), and [Chapter 7](#)’s string `join` method (which makes a new string by putting a substring between strings in an iterable). Hence, these will also work on an open file and automatically read one line at a time:

```
>>> list(open('data.txt'))
['Testing file IO\n', 'Learning Python, 6E\n', 'Python 3.12\n']

>>> tuple(open('data.txt'))
('Testing file IO\n', 'Learning Python, 6E\n', 'Python 3.12\n')

>>> '&&'.join(open('data.txt'))
'Testing file IO\n&&Learning Python, 6E\n&&Python 3.12\n'
```

Even some tools you might not expect fall into this category. For example, [Chapter 11](#)’s sequence assignment (original and starred), the `in` membership test, slice assignment, the list’s `extend` method, and single-star literal unpacking also leverage the iteration protocol to scan, and thus read a file by lines automatically:

```
>>> a, b, c = open('data.txt')           # Sequence assignment
>>> b, c
('Learning Python, 6E\n', 'Python 3.12\n')

>>> a, *b = open('data.txt')             # Extended-unpacking assignment
>>> a, b
('Testing file IO\n', ['Learning Python, 6E\n', 'Python 3.12\n'])

>>> 'Python 2.7\n' in open('data.txt')    # Membership test
False
>>> 'Python 3.12\n' in open('data.txt')
True

>>> L = [11, 22, 33, 44]                  # Slice assignment
>>> L[1:3] = open('data.txt')
>>> L
[11, 'Testing file IO\n', 'Learning Python, 6E\n', 'Python 3.12\n', 44]

>>> L = [11]
>>> L.extend(open('data.txt'))            # list.extend method
>>> L
[11, 'Testing file IO\n', 'Learning Python, 6E\n', 'Python 3.12\n']

>>> L = [11, *open('data.txt'), 44]       # List-literal unpacking
>>> L
[11, 'Testing file IO\n', 'Learning Python, 6E\n', 'Python 3.12\n', 44]
```

Remember that `extend` iterates automatically, but `append` does not—though you can use the latter to add an iterable to a list without iterating, and iterate across it later:

```
>>> L = [11]
>>> L.append(open('data.txt'))
>>> list(L[-1])
['Testing file IO\n', 'Learning Python, 6E\n', 'Python 3.12\n']
```

Nor does the iteration grand finale end here. In the prior chapter, we saw that the built-in `dict` call accepts an iterable zip result too. For that matter, so does the `set` call, as well as the set and dictionary comprehension expressions we met earlier and will revisit later:

```
>>> set(open('data.txt'))
{'Python 3.12\n', 'Learning Python, 6E\n', 'Testing file IO\n'}

>>> {line for line in open('data.txt')}
{'Python 3.12\n', 'Learning Python, 6E\n', 'Testing file IO\n'}

>>> {ix: line for ix, line in enumerate(open('data.txt'))}
{0: 'Testing file IO\n', 1: 'Learning Python, 6E\n', 2: 'Python 3.12\n'}
```

As noted, both set and dictionary comprehensions support the extended syntax of list comprehensions we met earlier in this chapter, including `if` tests:

```
>>> {line for line in open('data.txt') if line[0] in 'LP'}
{'Python 3.12\n', 'Learning Python, 6E\n'}

>>> {ix: line for (ix, line) in enumerate(open('data.txt')) if line[0] in 'LP'}
{1: 'Learning Python, 6E\n', 2: 'Python 3.12\n'}
```

Like the list comprehension, both of these scan the file line by line and pick out lines that begin with the specific letters. They also happen to build sets and dictionaries in the end, but we get a lot of work “for free” by combining file iteration and comprehension syntax. In the next part of this book, you’ll meet a relative of comprehensions—*generator expressions*—that deploys the same syntax and works on iterables too, but is also iterable itself:

```
>>> list(line.upper() for line in open('data.txt'))
['TESTING FILE IO\n', 'LEARNING PYTHON, 6E\n', 'PYTHON 3.12\n']
```

Other built-in functions support the iteration protocol as well, but frankly, some are harder to cast in interesting examples related to files. For example, the `sum` call computes the sum of all the numbers in any iterable; the `any` and `all` built-ins return `True` if any or all items in an iterable are `True`, respectively; and `max` and `min` return the largest and smallest item in an iterable, respectively. Like `reduce`, all of the tools in the following examples accept any iterable as an argument and use the iteration protocol to scan it, but return a single result:

```
>>> sum(range(5))           # Punt (requires numbers)
10
>>> any(open('data.txt'))   # Any/all lines true (nonempty)
True
>>> all(open('data.txt'))   # Mostly pointless for files
True
>>> max(open('data.txt'))    # Line with highest string value
'Testing file IO\n'
>>> min(open('data.txt'))    # Use cases wanted!
'Learning Python, 6E\n'
```

There’s one last iteration tool worth mentioning, although it’s a preview of this book’s next part: in [Chapter 18](#), you’ll learn that a special `*` form can be used in function calls to unpack a collection of values into individual arguments, much as it does in list (and other) literals. As you can probably predict by now, this accepts any iterable too:

```
>>> def f(a, b, c):                                # See Part IV
    print(a, b, c, sep='&')

>>> f(*open('data.txt'))                            # Iterates by lines too!
Testing file IO
&Learning Python, 6E
&Python 3.12
```

In fact, because this argument-unpacking syntax in calls accepts iterables, it's also possible to use the `zip` built-in to *unzip* zipped tuples, by making prior or nested `zip` results arguments for another `zip` call (warning: you probably shouldn't read the following example if you plan to operate heavy machinery anytime soon!):

```
>>> X, Y = (1, 2), (3, 4)
>>> list(zip(X, Y))                                # Zip tuples: returns an iterable
[(1, 3), (2, 4)]

>>> A, B = zip(*zip(X, Y))                          # Unzip a zip, really!
>>> A, B
((1, 2), (3, 4))
```

And that concludes our iteration-tools finale. It's probably not complete, but you probably get the point.

Other Iteration Topics

As mentioned in this chapter's introduction, there is more coverage of both list comprehensions and iterables in [Chapter 20](#), in conjunction with functions, and again in [Chapter 30](#) when we study classes. As you'll see later:

- User-defined functions can be turned into iterable *generator functions*, with `yield` statements.
- List comprehensions morph into iterable *generator expressions* when coded in parentheses.
- User-defined classes are made iterable with `__iter__` or `__getitem__` in *operator overloading*.

In particular, user-defined iterables defined with classes allow arbitrary objects and operations to be used in any of the iteration tools we've met in this chapter. By supporting just a single operation—*iteration*—objects may be used in a wide variety of contexts and tools.

Chapter Summary

In this chapter, we explored concepts related to looping in Python. We took our first substantial look at the *iteration protocol* in Python—a way for nonsequence objects to take part in iteration loops—and at *list comprehensions*. As we saw, a list comprehension is an expression similar to a `for` loop that applies another expression to all the items in any iterable object. Along the way, we also saw many of the other built-in iteration tools in Python's arsenal.

This wraps up our tour of specific procedural statements and related tools. The next chapter closes out this part of the book by discussing documentation options for Python code. Though a bit of a diversion from the more detailed aspects of coding, documentation is also part of the general syntax model, and it's an important component of well-written programs. In the next chapter, we'll also dig into a set of exercises for this part of the book before we turn our attention to larger structures such as functions. As usual, though, let's first exercise what we've learned here with a quiz.

Test Your Knowledge: Quiz

1. How are for loops and iterable objects related?
2. How are for loops and list comprehensions related?
3. Name four iteration tools in the Python language.
4. What is the best way to read line by line from a text file today?

Test Your Knowledge: Answers

1. The for loop normally uses the *iteration protocol* to step through items in the iterable object across which it is iterating. It first fetches an iterator from the iterable by calling the iterable's `__iter__`, and then calls this iterator object's `__next__` method on each iteration to advance and catches its `StopIteration` exception to determine when to stop looping. Any object that supports this model works in a for loop and in all other iteration tools. The protocol's methods can also be run manually with built-ins `iter` and `next`. For some objects that are their own iterator, the initial `iter` call is extraneous but harmless.
2. Both are iteration *tools*. List comprehensions are a concise and often efficient way to perform a common for loop task: collecting the results of applying an expression to all items in an iterable object. It's always possible to translate a list comprehension to a for loop, and part of the list comprehension expression looks like the header of a for loop syntactically. The for loop, however, can be used in additional looping roles that comprehensions do not address.
3. Iteration tools in Python include the for loop; list comprehensions; the `map` built-in function; the `in` membership test expression; and the built-in functions `sorted`, `sum`, `any`, and `all`. This category also includes the `list` and `tuple` built-ins, string `join` methods, and sequence assignments (starred or not), all of which use the iteration protocol (see answer #1) to step across iterable objects one item at a time.
4. The “best” way to read lines from a text file today is to not read it explicitly at all: instead, open the file within an iteration tool such as a for loop or list comprehension, and let the iteration tool automatically scan one line at a time by running the file's `next` handler method on each iteration. This approach is generally best in terms of coding simplicity, memory space, and possibly execution speed requirements.

The Documentation Interlude

This part of the book concludes with a brief look at techniques and tools used for documenting Python code. Although Python code is comparatively readable by itself, a few well-placed and human-accessible comments can do much to help others understand the workings of your programs—especially when code grows larger than most in this book. As you’ll see, Python includes both syntax and tools to make documentation easier. In particular, the *Pydoc* system covered here can render a code file’s documentation as either plain text in an interactive REPL, or HTML in a browser.

While this topic is partly tools related, it’s presented here because it both involves Python’s syntax model and provides a resource for readers struggling to understand Python’s toolset. For the latter purpose, this chapter also expands on documentation pointers first given in [Chapter 4](#). As usual, because this chapter closes out its part, it also ends with some warnings about common pitfalls and a set of exercises for this part of the text, in addition to its chapter quiz.

Python Documentation Sources

By this point in the book, you’re probably starting to realize that Python comes with an amazing amount of prebuilt functionality—built-in functions and exceptions, predefined object attributes and methods, standard-library modules, and more. And we’ve really only scratched the surface of each of these categories.

One of the first questions that bewildered beginners often ask is, How do I find information on all the built-in tools? This section provides tips on the various documentation sources available in Python. It also presents documentation strings (*docstrings*, for short), and the *Pydoc* system shipped with Python that makes use of them. These topics are somewhat peripheral to the core language itself, but they become essential knowledge as soon as your code becomes large or complex enough to challenge its readers—including yourself, six months down the road.

As summarized in [Table 15-1](#), there are a variety of places to look for information about Python, with generally increasing verbosity. Because documentation is such a crucial tool in practical programming, we’ll explore each of these categories in the sections that follow.

Table 15-1. Python documentation sources

Form	Role
# comments	In-file documentation
The <code>dir</code> function	Lists of attributes available in objects
Docstrings: <code>__doc__</code>	In-file documentation attached to objects
Pydoc: the <code>help</code> function	Interactive help for objects
Pydoc: HTML reports	Module documentation in a browser
Sphinx third-party tool	Richer documentation for larger projects
The standard manuals	Official language and library descriptions
Web resources	Online tutorials, examples, and so on

Comments

As we’ve learned—and used in most example listings so far—hash-mark comments are the most basic way to document your code. Python simply ignores all the text following a `#` (as long as it’s not inside a string literal), so you can follow this character with any words and descriptions meaningful to programmers. Such comments are accessible only in your source files, though; to code comments that are more widely available, you’ll use docstrings (ahead).

In fact, current *best practice* (generally accepted convention) dictates that docstrings are best for larger functional documentation (e.g., “my script does this”), and `#` comments are best limited to smaller code documentation (e.g., “this strange expression does that”) and are best limited in scope to a statement or small group of statements within a script or function. Docstrings are a broader topic we’ll get to in a moment; first, let’s see how to explore objects.

The `dir` Function

As we’ve also seen, the built-in `dir` function is an easy and automatic way to list all of the *attributes* available inside an object (i.e., its methods and simpler data items). It can be called with no arguments to list variables in the caller’s scope, something the next part of this book will define in full. More usefully, it can also be called with any object that has attributes, including imported modules and objects of built-in types, as well as the name of a data type itself. For example, to find out what’s available in a *module*, such as the standard library’s `sys`, import it and pass it to `dir`:

```
$ python3
>>> import sys
>>> dir(sys)
[...many names here...]
```

The list of attribute name strings you’ll get back is omitted here because it’s not small and is prone to vary per Python version; run this on your own for a better look. In fact, there are currently 110 attributes in `sys`, though we generally care only about the 97 that do not have leading double underscores (two underscores usually means interpreter-related), or the 83 that have no leading underscore at all (one underscore usually means implementation private, informally). Rooting out such numbers is a prime example of the preceding chapter’s list comprehension at work:

```
>>> len(dir(sys))                                # Number names in sys
110
>>> len([x for x in dir(sys) if not x.startswith('__')]) # Non __X names only
97
```



```
>>> len([x for x in dir(sys) if not x[0] == '_'])           # Ignore _X names too
83
```

To find out what attributes are provided in objects of *built-in types*, run `dir` on a literal, an existing instance, or the name of the desired type. For example, to see all text-string attributes, you can pass an empty or the type name `str`:

```
>>> dir('')
['__add__', ...more names here..., 'zfill']

>>> dir(str) == dir('')
True
```

The `dir` result for any built-in type includes a set of attributes that are related to the implementation of that type (mostly, for expression operators). Much as in modules, they all begin and end with double underscores to make them distinct, and you can safely ignore them at this point in the book (they'll resurface later for OOP). For instance, there are 81 text-string attributes, but only 47 that correspond to named methods or data; lists have even fewer:

```
>>> len(dir('')), len([x for x in dir('') if not x.startswith('__')])
(81, 47)
>>> len(dir([])), len([x for x in dir([]) if not x.startswith('__')])
(48, 11)
```

To list names, filter out double-underscored items that are not of common program interest by running the same list comprehensions, but print the attributes. To demo, here are the named attributes in lists and dictionaries:

```
>>> [a for a in dir(list) if not a.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']

>>> [a for a in dir(dict) if not a.startswith('__')]
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
```

This may seem like a lot to type to get an attribute list, but beginning in the next chapter, you'll learn how to wrap such code in a simple, importable, and reusable *function* so you don't need to type it again.

Aside: type names like `str`, `list`, and `dict` work in `dir` because they are actually names of types in Python today, not just type-converter functions; calling one of these invokes its constructor to generate an instance of that type. **Part VI** will have more to say about constructors and operator overloading methods when we discuss classes.

The `dir` function is mostly a memory jogger—it provides a list of attribute names, but it does not tell you anything about what those names mean. For such extra information, we need to move on to the next documentation source.



Attributes in IDEs: Some GUIs for Python work, including IDLE, have features that list attributes on objects automatically, which can be viewed as alternatives to `dir`. IDLE, for example, will list an object's attributes in a pop-up selection window when you type a period after the object's name and pause or press Tab. This is mostly meant as an *autocomplete* feature, though, not an information source. **Chapter 3** and **Appendix A** have introductory info on IDLE.

Docstrings and `__doc__`

Besides `#` comments, Python supports documentation that is automatically attached to objects and retained at runtime for inspection. Syntactically, such comments are coded as *string literals* of any type, except bytes and f-strings. They are located at the tops of code *components* (module files and function and class statements), and before any other executable code (`#` comments, including Unix-style `#!` lines, are OK before them). When present, the text of such strings, known as *docstrings*, is automatically stuffed into the `__doc__` attributes of the corresponding objects.

User-defined docstrings

For example, consider the file *docstrings.py* in [Example 15-1](#). You can safely ignore most of its code here (we'll study functions, modules, and classes in the next three parts of this book), but notice its three string literals: because they are coded at the beginning of the file, and at the start of a function (`def`) and class (`class`) within it, they are taken as docstrings and are saved in the associated object's `__doc__` when this file is loaded by a run or import.

Example 15-1. docstrings.py

```
"""
Module documentation
This module defines a name, function and class.
"""

edition = 6

def square(x):
    r'''
    Function documentation
    Returns the \square\ of its \numeric\ argument.
    '''
    return x ** 2 # power operator

class PartVI:
    "Class documentation for \U0001F40D 🍌"
    pass

# Top-level code
print(square(edition))
print(square.__doc__)
print(PartVI.__doc__)
```

Some fine points about this example's docstrings:

- *Literals*: the first of this file's docstrings uses a *triple-quoted* block at the top of the file. This is common for docstrings as it allows for multiline comments, but any sort of text string (except f-strings) will work. Both single- or double-quoted one-liners like that in the class are fine, but don't readily support multiline text.
- *Raw strings*: as the function demos, `r'...'` strings work too—and may even be required to suppress unwanted backslash escapes and avoid a syntax warning (and eventual *error*!) for unrecognized escapes each time code is run or recompiled to bytecode (see [Chapter 7](#)'s coverage of new `\` deprecations in Python 3.12).

- *Content*: docstrings can contain any sort of text, including the class's *Unicode* escape and raw emoji (per the Unicode intro in [Chapter 4](#) and the whole story in [Chapter 37](#)). Notice that the `\U` Unicode escape requires backslashes to be used, so it precludes using raw strings to avoid syntax errors in the future (use `\\`).

The whole point of this documentation protocol is that your comments are *retained* for inspection in `__doc__` attributes after the file is loaded. Thus, to display the docstrings associated with the module and the two code components it defines, we simply *import* the file and print their `__doc__` attributes, where Python has saved the text. Assuming this module is located in the directory in which we're currently working (and deferring to [Chapter 3](#)'s note about imports and directories for background info on why that matters):

```
$ python3
>>> import docstrings
36

    Function documentation
    Returns the \square\ of its \numeric\ argument.

Class documentation for 🍷👍
>>> print(docstrings.__doc__)

Module documentation
This module defines a name, function and class.

>>> print(docstrings.square.__doc__)

    Function documentation
    Returns the \square\ of its \numeric\ argument.

>>> print(docstrings.PartVI.__doc__)
Class documentation for 🍷👍
```

You will usually want to use `print` like this to view docstrings; otherwise, you'll get a single string with embedded `\n` newline characters. Also note that `import` runs the file's top-level prints (hence the output immediately following the `import`), as does launching it as a top-level script. Within the file, `square` is just `square`, a simple variable:

```
$ python3 docstrings.py
36

    Function documentation
    Returns the \square\ of its \numeric\ argument.

Class documentation for 🍷👍
```

You can also attach docstrings to *methods* of classes (covered in [Part VI](#) too), but because, as you'll learn, these are just `def` statements nested in `class` statements, they're not a special case. For instance, to fetch the docstring of a method function inside a class within a module, you would simply extend the path to go through the class: `module.class.method.__doc__`. We'll code an example of method docstrings in [Chapter 29](#).



Docstrings futurism: If you look closely, you'll notice that multiline docstrings, like that of `square`, retain their leading *indentation* as it was in the code. Python 3.13, still on the drawing board as this note is being written, plans to remove leading indentation from docstrings, mostly to save space in bytecode files and memory. Even before this mod, though, you won't normally see the indents in docs, because tools like `Pydoc` already strip and reformat, as you'll learn shortly.

Docstring standards

As mentioned earlier, common practice today recommends hash-mark comments for only smaller-scale documentation about an expression, statement, or small group of statements. Docstrings are better used for higher-level and broader functional documentation for a file, function, or class, and have become an expected part of Python software. Beyond these guidelines, though, you still must decide what to write.

Although some companies have internal standards, there is no broad consensus about what should go into the text of a docstring. There have been various markup language and template proposals, including HTML, but they don't seem to have caught on in the Python world. Frankly, convincing programmers to document their work using handcoded HTML probably won't happen in our lifetimes, but this shouldn't apply to documentation in general.

Documentation tends to have a lower priority than it should. Too often, if you get any docs in a file at all, you count yourself lucky (and even better if they're accurate and up to date). While technically optional, documentation is a crucial part of well-written programs, and you're encouraged to comment your code liberally. When you do, though, there is no standard docstring format; if you want to use them, anything goes today. Just as for writing code itself, it's up to you to create docstring content and keep it up to date, but common sense is probably your best ally in both tasks.

Built-in docstrings

While it's important to document our own code, it turns out that built-in modules and objects in Python use similar techniques to attach documentation above and beyond the attribute lists returned by `dir`. For example, to view a human-readable description of a built-in module, import it and print its `__doc__` string—as we did for our own code:

```
>>> import sys
>>> print(sys.__doc__)
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.
```

Dynamic objects:

```
argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
...more text omitted...
```

For finer-grained details, functions, classes, and methods within built-in modules have attached descriptions in their `__doc__` attributes as well:

```
>>> print(sys.getrefcount.__doc__)
Return the reference count of object.
The count returned is generally one higher than you might expect,
because it includes the (temporary) reference as an argument to
getrefcount().
```

You can also read about built-in functions via their docstrings:

```
>>> print(map.__doc__)
map(func, *iterables) --> map object
Make an iterator that computes the function using arguments from
each of the iterables. Stops when the shortest iterable is exhausted.
```

In fact, you can get a wealth of information about built-in tools by inspecting their docstrings this way—but you don’t have to. The Python `help` function largely automates this for you, as the next section will explain.

Pydoc: The help Function

The docstring technique proved to be so useful that Python eventually added a tool that makes docstrings even easier to display. Namely, *Pydoc* is Python code that knows how to extract docstrings and associated structural information and format them into nicely arranged reports of various types. Additional tools for extracting and formatting docstrings are available in the open source domain (including third-party tools that may support structured text—search the Web for pointers), but Python ships with Pydoc in its always-present standard library.

There are a variety of ways to launch Pydoc, but the two most prominent are the built-in `help` function and the Pydoc browser-based interface. Of these, `help` may be the most straightforward to use and is always available in a REPL.

We met `help` briefly in [Chapter 4](#). It invokes Pydoc to generate a simple plain-text report for any Python object. In this mode, `help` text looks much like a “manpage” on Unix-like systems. In fact, outside GUIs like IDLE, `help` text works the same way as a Unix “more” when there are multiple pages of text—press the space bar to move to the next page, Enter (or your keyboard’s equivalent) to go to the next line, and Q to quit the display. GUIs generally scroll `help` text instead. Let’s turn to a few examples to see how this works.

Running `help` on built-in tools

Because built-in tools come with the docstrings we saw earlier, `help` gives us a level of reference docs for Python, midway between `dir` lists and the full manuals. Here it is live, reporting on a standard-library module’s function:

```
>>> import sys
>>> help(sys.getrefcount)
Help on built-in function getrefcount in module sys:

getrefcount(object, /)
    Return the reference count of object.

    The count returned is generally one higher than you might expect,
    because it includes the (temporary) reference as an argument to
    getrefcount().
```

You do not have to import `sys` in order to call `help`, but you generally have to import `sys` to get `help` on `sys` this way, because `help` expects an object reference to be passed in. Alternatively, you can also get `help` for a module you have not imported by quoting the module’s name as a string. For example, `help('sys')` and `help('sys.getrefcount')` both work too, without the import step.

For larger objects such as modules and classes, the help display is broken down into multiple sections, the preambles of which are shown here. Run this interactively to see the full report:

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

MODULE REFERENCE
    https://docs.python.org/3.12/library/sys.html
    ...more omitted...

DESCRIPTION
    This module provides access to some objects used or maintained by the
    ...more omitted...

SUBMODULES
    monitoring

FUNCTIONS
    __breakpointhook__ = breakpointhook(...)
    ...more omitted...

DATA
    __stderr__ = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf...
    ...more omitted...

FILE
    (built-in)
```

Some of the information in this report is docstrings, and some of it (e.g., function call patterns) is structural information that Pydoc gleans automatically by inspecting objects' internals using tools available to your code too.

Besides modules, you can also use help on built-in functions, methods, and types. To get help for a built-in type, try either the type name (e.g., dict for dictionary or str for string); an actual object of the type (e.g., {} or ''); or a method of an actual object or type name (e.g., 's'.join or str.join).¹ For an entire type, you'll get a large display that describes all the methods available for that type; for a method, the help is more focused:

```
>>> help(dict)
Help on class dict in module builtins:
class dict(object)
| dict() -> new empty dictionary
| dict(mapping) -> new dictionary initialized from a mapping object's
| ...more omitted...

>>> help(str.replace)
Help on method_descriptor:
replace(self, old, new, count=-1, /)
```

¹ Note that asking for help on an actual *string object* directly (e.g., help('xyz')) doesn't work as you may expect: you usually get no help, because strings are interpreted specially—as a request for help on a topic or tool by name, as described earlier. You must use the str type name in this context, though both an empty string (help('')) and string method names referenced through actual objects (help('').join) work fine. An *interactive* help mode, started by typing just help(), avoids manual help calls, and may avoid some of their drama.

```
Return a copy with all occurrences of substring old replaced by new.  
...more omitted...
```

```
>>> help('').replace)
```

```
Help on built-in function replace:
```

```
replace(old, new, count=-1, /) method of builtins.str instance
```

```
Return a copy with all occurrences of substring old replaced by new.  
...more omitted...
```

```
>>> help(ord)
```

```
Help on built-in function ord in module builtins:
```

```
ord(c, /)
```

```
Return the Unicode code point for a one-character string.
```

The “/” in function docs means that arguments before it must be passed by *position only* (not by name), but this is substantially above this chapter’s pay grade. See the next part of this book for details on this function syntax.

Running help on your own code

Finally, the help function works just as well on your modules as it does on built-ins. Here it is reporting on components defined in the *docstrings.py* file we coded earlier in [Example 15-1](#). Again, some of this is our docstrings, and some is information automatically extracted by inspecting objects’ structures:

```
>>> import docstrings
```

```
>>> help(docstrings.square)
```

```
Help on function square in module docstrings:
```

```
square(x)
```

```
Function documentation
```

```
Returns the \square\ of its \numeric\ argument.
```

```
>>> help(docstrings.PartVI)
```

```
Help on class PartVI in module docstrings:
```

```
class PartVI(builtins.object)
```

```
| Class documentation for 🍷👍
```

```
|
```

```
| Data descriptors defined here:
```

```
| ...more omitted...
```

And asking for help on the entire module collects all parts at once, with your docstrings in the mix:

```
>>> help(docstrings)
```

```
Help on module docstrings:
```

```
NAME
```

```
docstrings
```

```
DESCRIPTION
```

```
Module documentation
```

```
This module defines a name, function and class.
```

```
CLASSES
```

```
builtins.object
```

```
PartVI
```

```
class PartVI(builtins.object)
```

```
| Class documentation for 🍷👍
```

```
|
```

```
| Data descriptors defined here:
```

```
| ...more omitted...
```

```

FUNCTIONS
    square(x)
        Function documentation
        Returns the \square\ of its \numeric\ argument.

DATA
    edition = 6

FILE
    /Users/me/MY-STUFF/Books/Dev/6E/LP6E/LP6E/Examples/Chapter15/docstrings.py

```

If you look closely, you’ll notice that we didn’t get our file’s printed output on `import` this time, just because it was imported earlier; as noted in [Chapter 3](#), imports run just once per file per session (more in [Part V](#)).

Pydoc: HTML Reports

The text displays of the help function are adequate in many contexts, especially at the interactive prompt. To readers who’ve grown accustomed to richer presentation mediums, though, they may seem a bit rudimentary. This section presents the HTML-based flavor of Pydoc, which renders module documentation more graphically for viewing in a web browser, and can even open one automatically for you. This browser interface scheme combines both search and display in a web page that communicates with an automatically started local server.

Using Pydoc’s browser interface

To view docs in a browser, you’ll usually launch Pydoc’s main script with a `pydoc -b` command line. This spawns two components running locally on your device: a documentation *server*, and a web-browser *client* that provides both page display and search input for content fetched from the server. A command line like the following starts the show:

```

$ python3 -m pydoc -b
Server ready at http://localhost:53965/
Server commands: [b]rowser, [q]uit
server> q

```

This command works on all PCs; it’s known to also work on Android devices in the Pydroid 3 app’s Terminal, but is spotty on phones in general. It uses the Python `-m` argument to locate Pydoc’s *module* file on the module-import search path (covered in [Part V](#) of this book), and run it as a top-level script. Per [Appendix A](#) and [Chapter 3](#), use `py` instead of `python3` in this command on Windows. On macOS and Linux, a `pydoc3 -b` command has the same effect, and on Windows the “Module Docs” entry in Python’s *Start* menu automatically runs Pydoc’s `-b` mode.

However you start it, Pydoc’s browser mode opens with its index page, captured in [Figure 15-1](#). The index page lists all the modules available for imports and includes search boxes (e.g., Get loads a named module’s docs), as well as topic guides and reserved-word info. You’ll see links to the docs of every module on your module search path, including the directory where Pydoc is launched. Pydoc’s server runs on the local host and on a dedicated but by default arbitrary unused port, but this can be tailored; launch Pydoc without `-b` for more options.

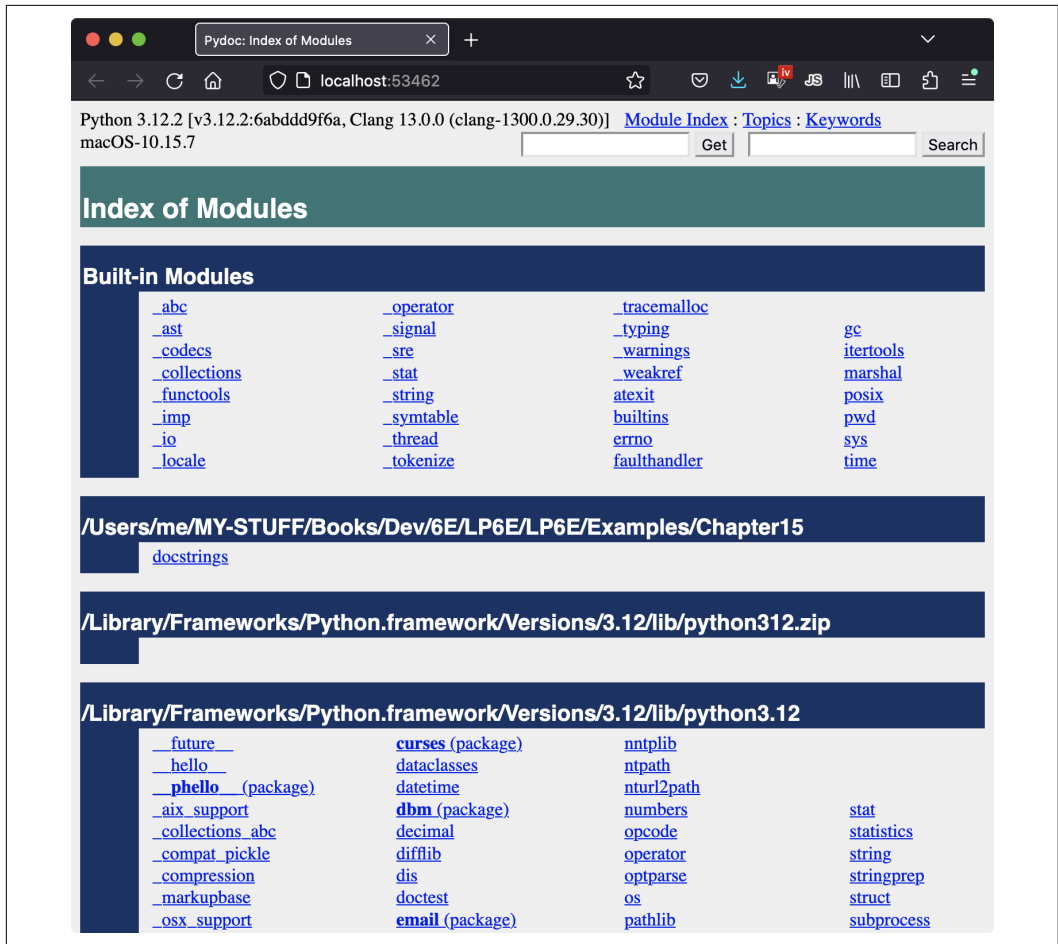


Figure 15-1. The top-level index start page of the Pydoc browser interface

When you click or tap on a module's entry in Pydoc's module-index page (or enter its name in Get), you get a page of docs for that module alone—which is essentially the same info that `help` provides in an interactive REPL, but formatted for display in your browser. Figure 15-2, for example, shows the page we get for the `docstrings` module we coded earlier in Example 15-1. It's the same report created for `help(docstrings)` with a simple HTML layout.

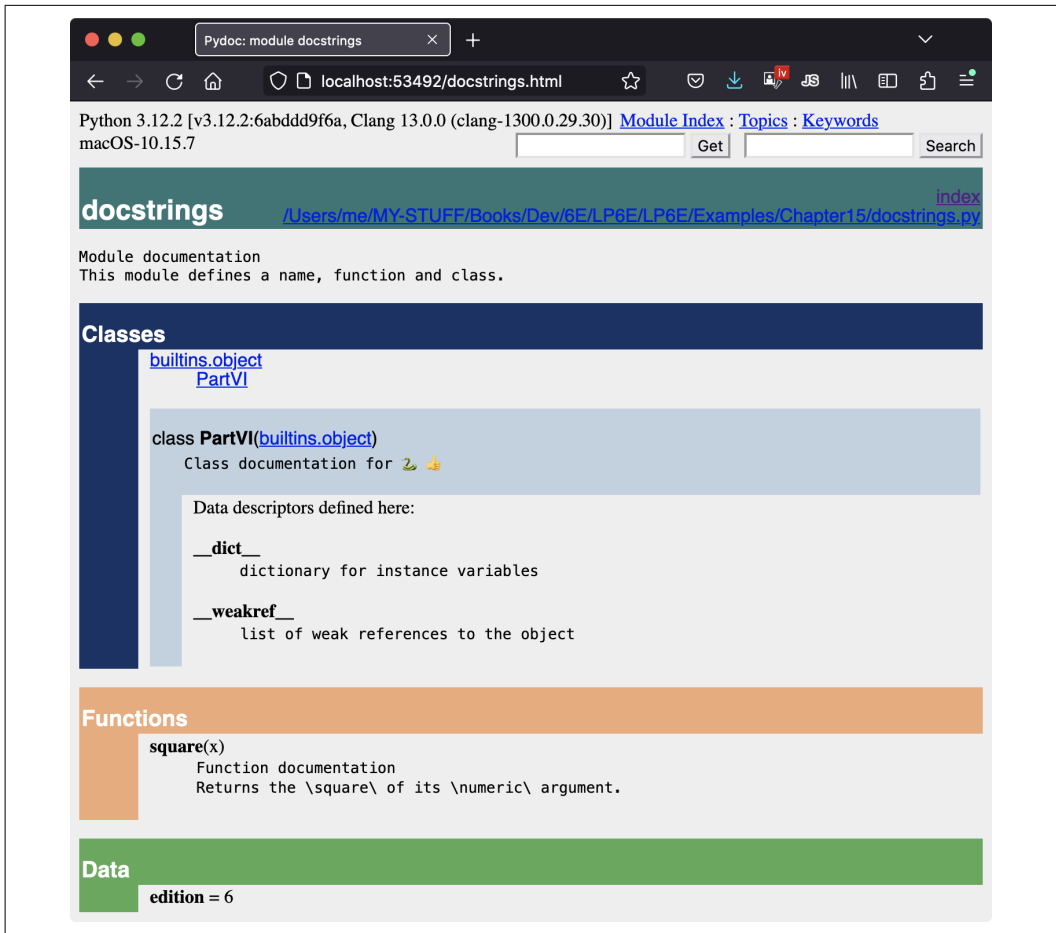


Figure 15-2. Pydoc browser interface displaying a user-defined module's docs

Experiment with Pydoc on your own for more insight. It's an optional tool, but provides extra docs for both built-in tools provided by Python, as well as docstring-loaded code files that you write yourself.

Customizing Pydoc

If you run this live, your Pydoc will probably look different than the screenshots in the book, because colors were customized on the machine used to capture the images. Though this is entirely optional and requires some web-development knowledge, you can customize Pydoc's appearance, too, by editing the simple CSS file it uses to render its pages. To modify colors, edit file `_pydoc.css` stored in the `pydoc_data` folder alongside Pydoc's module—which you can locate on any platform by importing Pydoc's module and inspecting its `__file__` attribute:

```
$ python3 -c "import pydoc; print(pydoc.__file__)"
...foldpath.../pydoc.py

$ ls ...foldpath.../pydoc_data          # Pydoc's CSS customization file is in here
__init__.py                            topics.py
__pycache__                            _pydoc.css
```

On Windows, use `py` and `dir` instead of `python3` and `ls`. This uses the Python `-c` command-line argument to submit code to be run as a string, which is handy for short bits of one-off code like this; here, the two statements separated by a semicolon run the same as they would if they were typed one at a time at a REPL's `>>>` prompt.

You can also find the path to Pydoc by running `help('pydoc')`: its path shows up at the bottom of the display. Once you've located the CSS file, edit it to customize as you like (but save the original first as a fallback). As an example, this book's captures use RGB color string `#173166` for `index-decor`. All of which assumes CSS skills beyond this book's scope (plus an undocumented config file that's prone to change or vanish in the future!), but is probably enough to get you started if you want to have a go.

More Pydoc tips

Pydoc works by *importing* selected files to extract their documentation. This has two usage implications. First, Pydoc can access only files in folders included on the module *search path* used by imports. This path automatically includes Python's standard library, so you can view docs for all built-in modules. It also includes the current working directory—the folder from which Pydoc was started—but that may be meaningless when started from a Windows Start button. Change your `PYTHONPATH` setting to include other code folders as needed, per [Appendix A](#).

Second, although Pydoc can render docs for both importable *modules* and runnable *scripts* in accessible folders, script docs comes with a catch: when a file is selected for help, Pydoc must import it in order to collect its documentation, and as we learned in [Chapter 3](#), importing *runs* a file's top-level code. Hence, script help implies a script run.

Imported modules normally just define tools when run, so this is usually irrelevant. If you ask for the documentation of a top-level script file, though, the script will be run, and the console window where you launched Pydoc serves as the script's standard input and output for any user interaction. This may work better for some scripts and modes than others; script IO may appear before or after help is dismissed in console mode, before help is scrolled in a GUI like IDLE, or interleaved oddly with Pydoc's own server-command prompts in browser mode.

Later in the book you'll learn ways to code top-level logic that's kicked off only when a file is run, not when it is imported (e.g., by nesting it under a test for variable `__name__` being `'__main__'`). If scripts use these protocols, they're safe to view in Pydoc, because it won't run any top-level code as an inadvertent side effect of viewing docs.

Finally, Pydoc has additional tools we'll skip here for space, including a `-w` switch for saving HTML docs to a file for later viewing, and a *plain-text* mode run from a command line with just a topic name that works the same as `help` in a REPL. Again, run Pydoc without any command-line arguments for its full set of options.



Blast from the past: Pydoc once had a simple GUI mode, invoked by flag `-g`, that launched a tkinter GUI client that communicated with the server. This was available until Python 3.2, when the current browser-only mode was deemed sufficient to warrant dropping the longstanding GUI mode in full. Browsers are GUIs too, of course, and the CSS configuration scheme for Pydoc that arose with the browser mode assumes extra web-development knowledge that many Python users won't have. In open source, those with time and desire to change things define the future for everyone.

Beyond Docstrings: Sphinx

If you're looking for a way to document your Python programs in a more sophisticated way, you may wish to check out *Sphinx*—a documentation generator system used to create Python's standard manuals described in the next section, as well as docs for many other software projects. It uses simple *reStructuredText* as its markup language and inherits much from the *Docutils* suite of parsing and translating tools.

Among other things, Sphinx supports a variety of output formats, automatic cross-references and indexes, and automatic code highlighting (colorization) using *Pygments*, which is itself a noteworthy Python tool. This is probably overkill for smaller programs where docstrings and `help` may suffice, but can yield higher-grade documentation for larger projects. See the web for more details on Sphinx, its related tools, and other options in the docs domain.

The Standard Manuals

As you know by now, this book is a *tutorial* that teaches by example. While its index and table of contents can be used to hunt for random topics after the fact, it's mainly designed to be read, not to serve as a reference resource. Given that, you'll probably want to supplement this book with reference tools once you move on to real projects.

Among these, Python's *standard manuals* may provide the most complete and up-to-date reference to the language and its toolset. You can easily view these manuals online with the Documentation link at Python's [website](#). They can also be viewed via the “Manuals” entry in Python's entry in the Start menu on Windows, and can be opened from “Python Docs” in the Help menu within the IDLE coding GUI. See your toolset for other access options.

When first opened, the manuals display a root page with a search box, like that captured in [Figure 15-3](#). The two most important entries here are most likely the “Library reference” (which documents built-in types, functions, exceptions, and standard-library modules) and the “Language reference” (which provides a formal description of language-level details). Both will probably be regular companions once you start coding Python in earnest.

Also of notable interest, the “What's new” documents in this standard manual set chronicle Python changes made in each release beginning with Python 2.0, which came out in late 2000—useful for those porting older Python code, or older Python skills. These documents are also useful for uncovering additional details on the differences in the Python 2.X and 3.X language lines, as well as recent Python 3.X changes covered in this book.

For both better and worse, change has been a constant in Python since its 0.X days. While the future is impossible to predict, the standard manuals' “What's new” docs will also cover Python mods almost certain to arise after this book's release. It's nearly required reading for anyone working downstream of a perpetually morphing sandbox.

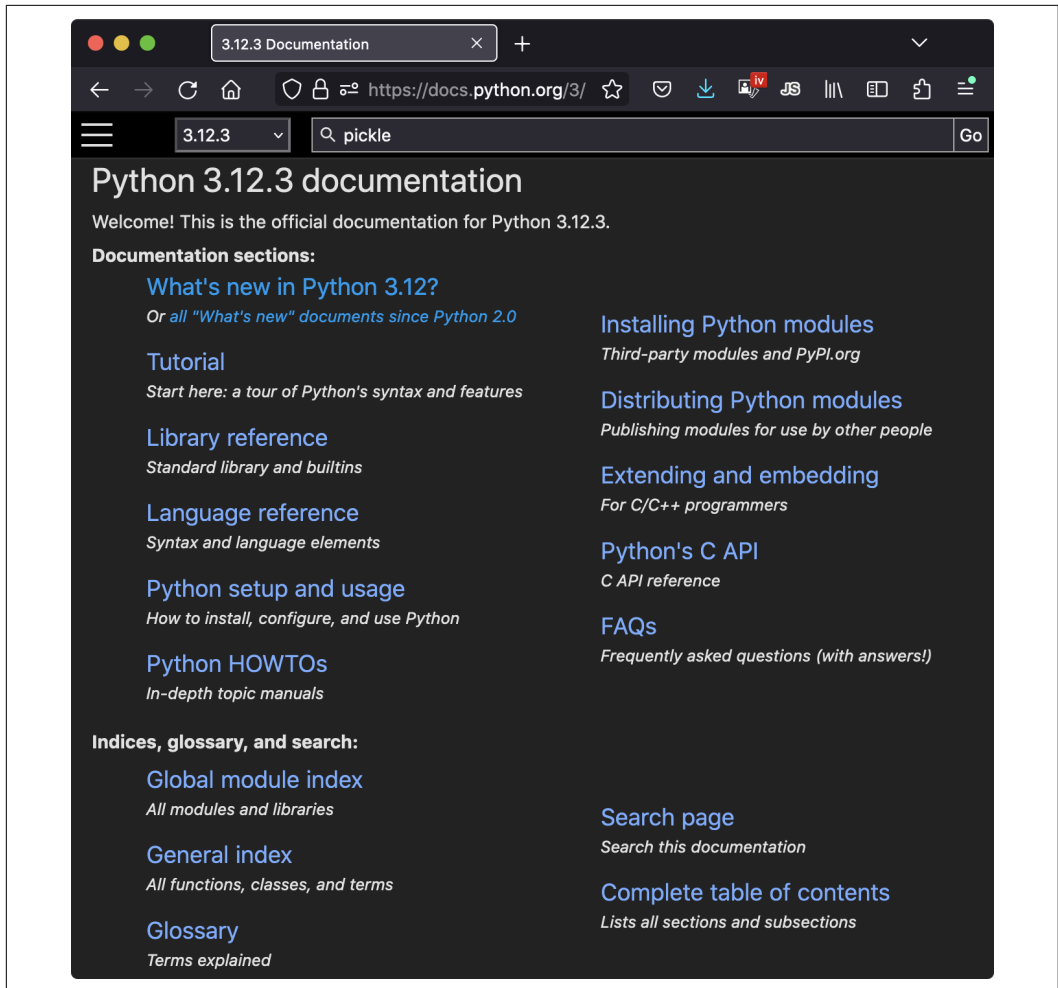


Figure 15-3. Python's standard manuals, available on the web and elsewhere

Web Resources

Finally, besides the standard manuals, Python's [website](#) also hosts additional resources, some of which cover special topics or domains, including non-English Python resources and introductions scaled to different target audiences.

Today you will also find a multitude of Python blogs, websites, wikis, and other resources on the web at large. Given the explosion of the web in recent decades, some online resources are naturally more authoritative and reliable than others, and some are sadly geared more toward monetization than education. This word of caution includes AI chatbots, which can only do as well as the data they're fed and paraphrase, and are not a replacement for the deep learning needed to work in software.

That said, there's a wealth of Python material out there to be had—for those willing to exercise the prudence and skepticism that today's web demands.

Common Coding Gotchas

Before the programming exercises for this part of the book, let's run through some of the most common mistakes beginners make when coding Python statements and programs. Many of these are warnings issued earlier in this part of the book, collected here for ease of reference. You'll learn to avoid these pitfalls once you've gained a bit of Python coding experience, but a few words now might help you avoid falling into some of these traps initially:

- **Don't forget the colons.** Always remember to type a `:` at the end of compound statement headers—the first line of an `if`, `match`, `while`, `for`, etc. You'll probably forget at first (as have thousands of Python students over the years), but you can take some comfort from the fact that it will soon become an unconscious habit.
- **Start in column 1.** Be sure to start top-level (unnested) code in column 1. That includes unnested code typed into module files, as well as unnested code typed at the interactive prompt (a.k.a. REPL). Indented means nested in Python, so it doesn't work at the top.
- **Blank lines matter at the interactive prompt.** Blank lines in compound statements are always irrelevant and ignored in module files, but when you're typing code at the interactive prompt, they end the statement. In other words, blank lines tell the interactive REPL that you've finished a compound statement; if you want to continue, don't hit the Enter key at the `...` prompt (if shown) until you're really done. This also means you can't paste multiline code at this prompt; it must run one full statement at a time, with blank lines as needed.
- **Indent consistently.** Avoid mixing tabs and spaces in the indentation of a nested block. Otherwise, what you see in your editor may not be what Python sees when it counts tabs as a number of spaces. This is true in any block-structured language, not just Python—if the next programmer has tabs set differently, it will be difficult or impossible to understand the structure of your code. It's safer to use all tabs or all spaces for each block.
- **Don't code C in Python.** A reminder for C/C++ programmers: you don't need to type parentheses around tests in `if` and `while` headers (e.g., `if (X==1)`). You can if you like (any expression can be enclosed in parentheses), but they are fully superfluous in this context. Also, do not terminate all your statements with semicolons; it's technically legal to do this in Python as well, but it's totally useless unless you're placing more than one statement on a single line (the end of a line normally terminates a statement). And remember, don't use `{}` around blocks (indent your nested code blocks consistently instead), and avoid embedding assignment statements in `while` loop tests unless they're simple (even though Python now enables this with `:=`).
- **Use simple for loops instead of while or range.** Another reminder: a simple `for` loop (e.g., `for x in y`) is almost always simpler to code and often quicker to run than a `while`- or `range`-based counter loop. Avoid the temptation to count things in Python; though occasionally required, it's usually subpar.
- **Beware of mutables in assignments.** As cautioned before, be careful about using mutables in a multiple-target assignment (`a = b = []`), as well as in an augmented assignment (`a += [1, 2]`). In both cases, in-place changes may impact other variables. See [Chapter 11](#) for details if you've forgotten why this is true.
- **Don't expect results from functions that change objects in place.** We encountered this nag earlier, too: in-place change operations like the `list.append` and `list.sort` methods presented in [Chapter 8](#) do not return values (other than `None`), so you should call them without assigning the result. It's not uncommon for beginners to say something like `mylist = mylist.append(X)` to try

to get the result of an `append`, but this assigns `mylist` to `None`, not to the modified list (in fact, you'll lose your reference to the list altogether).

- **Always use parentheses to call a function.** You must add parentheses after a function name to call it, whether it takes arguments or not (e.g., use `function()`, not `function`). In the next part of this book, you'll learn that functions are simply objects that have a special operation—a call that you trigger with the parentheses—but they can be referenced like any other object without triggering a call. This often crops up with files; it's common for beginners to type `file.close` to close a file, rather than `file.close()`. Because it's legal to reference a function without calling it, the first version succeeds silently, but it does not close the file.
- **Don't use extensions or paths in imports and reloads.** Omit directory paths and file extensions in `import` statements—say `import mod`, not `import mod.py`. We discussed module basics in [Chapter 3](#) and will continue studying modules in [Part V](#). Because modules may have other extensions besides `.py` (e.g., `.pyc` for bytecode), hardcoding a particular extension is not only invalid syntax; it doesn't make sense. Python picks an extension automatically, and any platform-specific directory path syntax comes from module search path settings, not the `import` statement. Until we explore this in more depth, use just a simple name in imports.
- **And other pitfalls in other parts.** Be sure to also see the built-in type warnings at the end of [Part II](#), as they may qualify as coding issues too. There are additional “gotchas” that crop up commonly in Python coding—losing a built-in function by reassigning its name, hiding a library module by using its name for one of your own, changing mutable argument defaults, and so on—but we don't have enough background to cover them yet. To learn more about both what you should and shouldn't do in Python, you'll have to read on; later parts extend the set of “gotchas” and fixes we've enumerated here.

Chapter Summary

This chapter toured documentation—both documentation we write ourselves in our own programs, and documentation available for tools we use. We met docstrings, explored reference resources for Python, and learned how Pydoc's help function and browser-based interfaces provide extra sources of documentation. Because this is the last chapter in this part of the book, we also reviewed common coding mistakes to help you avoid them.

In the next part of this book, we'll start applying what we already know to larger program constructs. Specifically, the next part takes up the topic of *functions*—a tool used to group statements for reuse. Before moving on, however, be sure to work through the set of lab exercises for this part of the book that appear at the end of this chapter. And even before that, let's run through this chapter's quiz to review.

Test Your Knowledge: Quiz

1. When should you use documentation strings instead of hash-mark comments?
2. Name three ways you can view documentation strings.
3. How can you obtain a list of the available attributes in an object?
4. How can you get a list of all importable modules on your computer?

Test Your Knowledge: Answers

1. Documentation strings (docstrings) are considered best for larger, functional documentation, describing the use of modules, functions, classes, and methods in your code. Hash-mark comments are better limited to smaller-scale documentation about arcane expressions or statements at strategic points on your code. This is partly because docstrings are easier to find in a source file (they have specific locations), but also because they can be extracted and displayed by the Pydoc system.
2. You can see docstrings by printing an object's `__doc__` attribute, by passing it to Pydoc's `help` function, and by selecting modules in Pydoc's HTML-based *browser* interface. The latter is launched with a `-b` command-line switch, and runs a client/server system that displays documentation in a popped-up web browser. Pydoc can also be run to save a module's documentation in an HTML file for later viewing or printing.
3. The built-in `dir(X)` function returns a list of all the attributes attached to any object. A list comprehension like `[a for a in dir(X) if not a.startswith('__')]` can be used to filter out internals' names with underscores (you'll learn how to wrap this in a function in the next part of the book to make it reusable).
4. Because the index page opened for Pydoc's browser-based `-b` mode displays every module on your import search path, this shows all the modules available for imports in your programs. This relies on configurable search-path settings that we'll cover more fully later in this book, but it includes both the current directory and Python's standard library by default. You can also write code of your own to achieve this using the same tools the Pydoc uses (it's just Python code, after all), but it's a significant manual task.

Test Your Knowledge: Part III Exercises

Now that you know how to code basic program logic, the following exercises will ask you to implement some simple tasks with statements. Much of the work is in exercise 4, which lets you explore coding alternatives. There are always many ways to arrange statements, and part of learning Python is learning which arrangements work better than others. You'll eventually gravitate naturally toward what experienced Python programmers call “best practice,” but best practice takes practice.

See “[Part III, Statements and Syntax](#)” on page 1165 in [Appendix B](#) for solutions to the following exercises:

1. *Coding basic loops*: This exercise asks you to experiment with `for` loops.
 - a. Write a `for` loop that prints the ASCII code point of each character in a string named `S`. Use the built-in function `ord(character)` to convert each character to its integer code point. This function technically returns a Unicode code point that may not fall in ASCII's range, but if you restrict its content to ASCII characters, you'll get back ASCII codes. (Test it interactively to see how it works, if needed.)
 - b. Next, change your loop to compute the *sum* of the ASCII code points of all the characters in a string.
 - c. Finally, modify your loop again to return a new list that *contains* the ASCII code points of each character in the string. Does the expression `map(ord, S)` have a similar effect? How about `[ord(c) for c in S]`? Why? (Hint: see [Chapter 14](#).)

2. *Coding basic selections:* Write Python `if` and `match` statements that print the first three month names of the year, given their relative numbers. For example, given 1, the output should be January, and for 3, it should be March (or whatever months are named in your locale). Then do the same by coding the choice with both a dictionary-key index and a list-offset index. How would you handle out-of-range month numbers?
3. *Backslash characters:* What happens on your machine when you type the following code interactively?

```
for i in range(50):
    print(f'hello {i}\a')
```

Beware that if it's run outside of some interfaces like IDLE, this example may beep at you, so you may not want to run it in a crowded room! IDLE ignores odd characters instead of beeping—spoiling much of the joke (see the backslash escape characters in [Table 7-2](#)).

4. *Sorting dictionaries:* In [Chapter 8](#), we saw that dictionaries are collections that store keys by *insertion* order only. Write a `for` loop that prints a dictionary's items in sorted (ascending) *key/value* order. (Hint: store keys in unordered fashion, and use the dictionary keys and list `sort` methods, or the `sorted` built-in function.)
5. *Program logic alternatives:* Consider the following code, which uses a `while` loop and found flag to search a list of powers of 2 for the value of 2 raised to the fifth power (32). It's stored in a module file called *power.py*:

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

found = False
i = 0
while not found and i < len(L):
    if 2 ** X == L[i]:
        found = True
    else:
        i = i+1

if found:
    print('at index', i)
else:
    print(X, 'not found')

$ python3 power.py
at index 5
```

As is, the example doesn't follow normal Python coding techniques. Follow the steps outlined here to improve it (for all the transformations, you may either type your code interactively or store it in a script file run from the system command line or other interface—using a file makes this exercise much easier):

- a. First, rewrite this code with a `while` loop `else` clause to eliminate the found flag and final `if` statement.
- b. Next, rewrite the example to use a `for` loop with an `else` clause, to eliminate the explicit list-indexing logic. (Hint: to get the index of an item, use the list `index` method—`L.index(X)` returns the offset of the first `X` in list `L`.)

- c. Next, remove the loop completely by rewriting the example with a simple `in` operator membership expression. (See [Chapter 8](#) for more details, or type this to test: `2 in [1, 2, 3]`.)
- d. Finally, use a `for` loop and the list `append` method to generate the powers-of-2 list (`L`) instead of hardcoding a list literal.

Deeper thoughts:

- a. Do you think it would improve performance to move the `2 ** x` expression outside the loops? How would you code that?
- b. As we saw in exercise 1, Python includes a `map(function, iterable)` tool that can generate a powers-of-2 list, too: `map(lambda x: 2 ** x, range(7))`. Try typing this code interactively; you'll meet `lambda` more formally in the next part of this book, especially in [Chapter 19](#). Would a list comprehension help here (see [Chapter 14](#))? How about a `:=` expression (see [Chapter 11](#))?

Functions and Generators

Function Basics

In **Part III**, we studied basic procedural statements in Python. Here, we'll move on to explore a set of additional statements and expressions that we can use to create functions of our own.

In simple terms, a *function* is a package of code invoked by name. It labels and groups a set of statements so they can be run more than once in a program. A function also can compute a result value, and lets us specify parameters that serve as inputs and may differ each time the function's code is run. Wrapping an operation in a function makes it a generally useful tool, which we can apply in a variety of contexts.

On a more pragmatic level, functions are the alternative to programming by *cutting and pasting*—rather than having multiple redundant copies of an operation's code, we can factor it into a single function. In so doing, we reduce our future work radically: if the operation must be changed later, we have only one copy to update in the function, not many scattered throughout the program.

Functions are also the most basic program structure Python provides for maximizing code *reuse*, and lead us to the larger notions of program *design*. As you'll see, functions let us split complex systems into manageable parts. By implementing each part as a function, we make it both reusable and easier to code.

Table 16-1 abstractly previews the function-related tools we'll study in this part of the book—a set that includes call expressions, two ways to make functions (`def` and `lambda`), two ways to manage scope visibility (`global` and `nonlocal`), two ways to send results back to callers (`return` and `yield`), and tools to pause for results (`async` and `await`). While this comprises a feature-rich topic, you'll find that its commonly used core is straightforward.

Table 16-1. Function-related statements and expressions

Statement or expression	Examples
Call expressions	<code>myfunc('hack', tool=python, *versions)</code>
<code>def</code>	<pre>def printer(message): print('Hello', message)</pre>
<code>return</code>	<pre>def adder(a, b=1, *c): return a + b + c[0]</pre>

Statement or expression	Examples
lambda	<code>funcs = [lambda x: x**2, lambda x: x**3]</code>
global	<code>x = 'old'</code> <code>def changer():</code> <code> global x; x = 'new'</code>
nonlocal	<code>def outer():</code> <code> x = 'old'</code> <code> def changer():</code> <code> nonlocal x; x = 'new'</code>
yield	<code>def squares(x):</code> <code> for i in range(x): yield i ** 2</code>
Generator expressions	<code>(i ** 2 for i in range(x))</code>
async/await	<code>async def consumer(a, b):</code> <code> await producer(b)</code>
Decorators and annotations	<code>@tracer</code> <code>def func(a: 'hack' = None) -> None</code>

Why Use Functions?

Before we get into the details, let's establish a clearer picture of what functions are all about. Functions are a nearly universal program-structuring device. You may have come across them before in other languages, where they may have been called *subroutines* or *procedures*. As a brief introduction, functions serve two primary development roles, and serve as the basis of other tools:

Maximizing reuse and minimizing redundancy

As in most programming languages, Python functions are the simplest way to package logic you may wish to use in more than one place and more than one time. Up until now, almost all the code we've been writing has run immediately. Functions allow us to defer and generalize code to be used arbitrarily many times later. Because they allow us to code an operation in a single place and use it in many others, functions are also a *factoring* tool: they enable us to reduce code redundancy in our programs, and thereby reduce maintenance effort.

Dividing and conquering

Functions also provide a tool for splitting systems into pieces that have well-defined roles and *manageable* scopes. For instance, to make a pizza from scratch, you would start by mixing the dough, rolling it out, adding toppings, baking it, and so on. If you were programming a pizza-making robot, functions would help you divide the overall “make pizza” task into smaller parts—one function for each subtask in the process. Because it's easier to implement the smaller tasks in isolation than it is to implement the entire process at once, this makes larger tasks more practical.

Implementing object methods

In general, functions are about *procedure*—how to do something, rather than what you're doing it to. Nevertheless, when paired with an implied subject, they can also be used to code object-specific behavior known as methods. You'll see why this distinction matters in [Part VI](#), when we start making new objects with classes. As you'll find, classes are largely just packages of the functions you'll learn to code here.

In this part of the book, we'll explore the tools used to code functions in Python: function basics, scope rules, and argument passing, along with a few related concepts such as generators, coroutines, and functional tools. Because its importance begins to become more apparent at this level of coding, we'll

also revisit the notion of polymorphism, which was introduced earlier in the book. As you'll see, functions don't imply much new syntax, but they do lead us to some bigger programming ideas.

Function Coding Overview

Although it wasn't made very formal, we've already *used* functions in earlier chapters. For instance, we called the built-in `open` function to make a file object, invoked the `len` built-in function to ask for the number of items in a collection object, and employed tools like `zip` and `range` for iteration tasks.

In this chapter, we will begin exploring how to write *new* functions in Python. Functions we write behave the same way as the built-ins we've already seen: they are called in expressions, are passed values, and return results. But writing new functions requires the application of a few additional tools and ideas that haven't yet been introduced. Moreover, functions behave very differently in Python than they do in compiled languages like C.

Basic Function Tools

As a road map and brief rundown on what we'll study in this part of the book, here are the basic components and concepts in Python's function toolbox:

- **def creates a function and assigns it to a name.** Python functions are coded with `def` statements. When Python reaches and runs a `def`, it generates a new function object and assigns it to the function's name. As with all assignments, the function name becomes a reference to the function object. There's nothing magical about the name of a function—as you'll see, the function object can be assigned to other names, stored in a list, and so on. Function objects may also have arbitrary user-defined *attributes* attached to them to record data.
- **def is executable code.** Unlike functions in compiled languages, `def` is an executable statement—your function does not exist until Python runs its `def`. In fact, it's legal (and even occasionally useful) to nest `def` statements inside `if` statements, `for` loops, and even other `defs`. In typical usage, `def` statements are coded at the top level of module files, and are automatically run to make functions when their modules are imported.
- **return sends a result object back to the caller.** When a function is called, the caller normally stops until the function finishes its work and returns control to the caller. Functions that compute a value send it back to the caller with a `return` statement, and the returned value becomes the result of the function call. A `return` without a value simply returns to the caller (and sends back `None`, the default result).
- **lambda creates a function but returns it as a result.** Function objects may also be created with the `lambda` expression, a feature that allows us to code *inline* function definitions in places where a `def` statement won't work syntactically. Like `def`, functions made this way are created when the `lambda` is reached and run. Because `lambda` is not typically coded at the top level of a module, though, such functions are made during program runs, not imports. `lambda` is an optional convenience best seen as a sidebar to the `def` statement.

Advanced Function Tools

In addition, Python functions can leverage more advanced tools like the following, which we'll take up after we've introduced the basics:

- **global declares module-level variables that are to be assigned.** By default, all names assigned in a function are local to that function and exist only while the function runs. To assign a name in the enclosing module, functions need to list it in a `global` statement. More generally, names are always looked up in *scopes*—places where variables are stored—and names are bound to scopes per the location of assignments in your code.
- **nonlocal declares enclosing-function variables that are to be assigned.** In the same category as `global`, the `nonlocal` statement allows a function to assign a name that exists in the scope of a syntactically enclosing `def` statement. This allows enclosing functions to serve as a place to retain *state*—information remembered between function calls—without using shared global names.
- **yield sends a result object back to the caller, but remembers where it left off.** Functions known as *generators* may also use the `yield` statement to send back a value and suspend their state such that they may be resumed later, to produce a series of results over time. Along with their *generator expression* kin, such functions save space and avoid delays just like the built-in iterables we met in the prior part of this book.
- **await/async pause a waiting function so that other tasks may run.** Functions known as *coroutines* can suspend their execution until a required result is available. This is an advanced applications-level topic, but allows code to use language tools to get other work done while waiting on a blocking event like slow IO.

General Function Concepts

Finally, along the way we'll explore a handful of core concepts that span functions of all kinds:

- **Arguments are passed by assignment (object reference).** In Python, arguments are passed to functions by assignment—which, as we've learned, means by object reference. As you'll see, in Python's model the caller and function share *objects* by references, but there is no name aliasing. Changing an argument name within a function does not also change the corresponding name in the caller, but changing passed-in mutable objects in place can change objects shared by the caller, and serve as a function result.
- **Arguments are passed by position, unless you say otherwise.** Values you pass in a function call match argument names in a function's definition from left to right by default. For flexibility, function *calls* can also pass arguments by name with `name=value` keyword syntax and unpack arbitrarily many arguments to send with `*args` and `**args` starred-value notation. Function *definitions* use the same syntax forms to specify argument defaults and collect arbitrarily many arguments received.
- **Arguments, return values, and variables are not declared.** As with everything in Python, there are no type constraints on functions. In fact, nothing about a function needs to be declared ahead of time: you can pass in arguments of any type, return any kind of object, and so on. As a consequence, a single function can often be applied to a variety of object types—any objects that have a compatible *interface* (support for expected methods and expressions) will do, regardless of their specific types. This makes code flexible by design.

- **Attributes, annotations, and decorators support advanced roles.** Functions can optionally be augmented with both user-defined attributes and other tools that extend their roles. Annotations of arguments and results, for example, can serve a variety of goals. Among these, *type hinting* (noted in [Chapter 6](#)) gives suggested object types but is unused by Python itself and simply serves as a weighty form of documentation, which we won't cover further in this book. More usefully, functions can also be prefixed with `@` decorators that add extra layers of logic but are used in ways that merit largely deferring until [Part VI](#) when we've also learned about classes.

If some of the preceding words didn't sink in, don't worry—we'll explore most of these concepts with real code in this part of the book. Let's get started by expanding on some of the basics by coding a few abstract examples.

def Statements

The `def` statement creates a function object and assigns it to a name. Its general format and usage is as follows:

```
def name(arg1, arg2,... argN):      # Define a function
    statements                     # Function body

name(val1, val2,... valN)          # Call it later in an expression
```

As with all compound statements, `def` consists of a *header* line followed by a block of other statements, usually indented (though a simple statement after the colon works as usual). The statement block becomes the function's *body*—that is, the code Python executes each time the function is later called.

The `def` header line specifies a function *name* that is assigned the new function object, along with a list of zero or more *arguments* (sometimes called *parameters*) enclosed in parentheses (even for zero arguments). The argument names in the header are *assigned* to the objects passed in parentheses at the point of call elsewhere in your code.

To *call* the function later in your program and run its body, you can use the function *name* assigned by `def`, passing in zero or more *values* (objects) to match the arguments listed in the `def` header. A call expression can also denote the function with other types of object references; it need not necessarily be the name coded in the `def` header.

return Statements

Function bodies often contain one or more `return` statements that make sense only inside a `def`:

```
def name(arg1, arg2,... argN):
    ...
    return result                # The result of the call expression
```

The Python `return` statement can show up anywhere in a function body. When reached, it *ends* the function call and sends a *result* back to the caller to serve as the result of the call expression. The `return` statement consists of an optional object-value expression that gives the result. If the value is omitted, `return` sends back a `None` by default.

The `return` statement itself is optional too; if it's not present, the function exits when the control flow falls off the end of the function body. Technically, a function *without* a `return` statement also returns the `None` object automatically, but this return value is usually ignored at the call, by using a call-expression *statement* of [Chapter 11](#).

Functions may also contain `yield` statements used to produce a series of values over time, as well as `await` and `async for/with` statements used to suspend the function’s execution, but we’ll defer discussion of these more advanced tools until we survey generator and coroutine topics in [Chapter 20](#).

def Executes at Runtime

The Python `def` is a true executable statement: when it runs, it creates a new function object and assigns it to a name. (Remember, all we have in standard Python is *runtime*; there is no such thing as a separate compile time.) Because it’s a statement, a `def` can appear anywhere a statement can—even *nested* in other statements. For instance, although `defs` are normally top-level code run when the module enclosing them is imported, it’s also completely legal to nest a function `def` inside an `if` statement to select between alternative definitions:

```
if test:
    def func():          # Define func this way
        ...
else:
    def func():          # Or else this way
        ...
...
func()                  # Call the version selected and built
```

One way to understand this code is to realize that the `def` is much like an `=` statement: it simply assigns a name (like `func` here) at runtime. Unlike in compiled languages such as C, Python functions do not need to be fully defined before the program runs. More generally, `defs` are not run until they are reached, and the code *inside* `defs` is not run until the functions are later called.

Because function definition happens at runtime, there’s nothing special about the function name. What’s important is the object to which it refers. For example:

```
othername = func        # Assign function object
othername()              # Call func again
```

Here, the function was assigned to a different name and called through the new name. Like everything else in Python, functions are just *objects*; they are recorded explicitly in memory at program execution time. In fact, besides calls, functions allow arbitrary *attributes* to be attached to record information for later use:

```
def func(): ...          # Create + assign function object
func()                  # Call object via its name
func.attr = value       # Attach attributes
```

In other words, functions are *first-class objects*, to borrow a term introduced in [Chapter 9](#). While they don’t support some operations that other objects do, functions inhabit the same category as every object. As you’ll see, passing them about and storing them in other objects is both syntactically legal and surprisingly useful.

lambda Makes Anonymous Functions

In addition to `def`, you can make a new function with the `lambda` expression. It’s coded and might be used like this:

```
name = lambda arg1, arg2,... argN: expression

name(val1, val2,... valN)
```

Like `def`, `lambda` makes a new function object to be called later. It begins with the word `lambda`, followed by an arguments-list *header* that works the same as in `def` but is coded without parentheses. Unlike `def`, the code after the `:` in `lambda` is a single *expression*—it cannot contain statements and is the *implied* body and return value of the function that `lambda` makes. We don't need to say `return` in a `lambda` (and we can't).

Also unlike `def`, `lambda` does not itself assign the new function to a name, but simply *returns* it as the result of the whole `lambda` expression. This snippet manually assigns the function to a name through which it is called, but that's optional: the result might also be saved in another object or passed for use elsewhere. Because of this, `lambda` is usually called an “anonymous” function—one that's unnamed.

You'll learn more about `lambda` later and see it in action along the way. But your first question may be, Why have a version of `def` whose code is limited to just one expression? In short, `lambda` can be used in places that `def` cannot, including in calls and object literals where we want to embed small bits of deferred and runnable code. While `def` handles larger tasks and also supports embedding functions by name, `lambda` is an optional amenity in simpler roles.

And if you're keeping track, we've now seen *four* expression equivalents for more general statements—the `lambda` for `def`, the `:=` named assignment for `=`, the `if/else` ternary for `if`, and the comprehension for `for`. Although lambdas are limited to expressions, they can use any of these expression equivalents to code assignments, logic, and loops. All these expression forms were accumulated over time, but are convenient alternatives in practical code.

A First Example: Definitions and Calls

Apart from their runtime flavor (which tends to seem most novel to programmers with backgrounds in compiled languages), Python functions are straightforward to use. Let's code a first real example to demonstrate the fundamentals, from both sides of the function equation: *definition* (the `def` or `lambda` that creates a function) and *calls* (the expressions that tell Python to run the function's body).

Definition

Here's a definition typed interactively that defines a function named `times`, which returns the product of its two arguments (it's not much, but it demos the basic bits). As usual “...” prompts are omitted here for copy and paste:

```
>>> def times(x, y):           # Create and assign function
    return x * y              # Body executed when called
```

When Python reaches and runs this `def`, it creates a new function object that packages the function's code, and assigns this object to the name `times`. Typically, such a statement is coded in a module file and runs when the enclosing file is imported; for something this small, though, the interactive REPL suffices.

As an aside, a `lambda` can have the same effect if we assign its result to a name, though it's typically used in more focused roles than this, and often not as top-level code. Try the `def`, `lambda`, or both if you're working along—they both make function objects that work the same:

```
>>> times = lambda x, y: x * y
```

Calls

Both `def` and `lambda` make a function but do not call it. After either has run, you can *call* (i.e., run) the function in your program by adding parentheses after the function's name. The parentheses may optionally contain one or more object arguments, to be passed (i.e., assigned) to the names in the function's header:

```
>>> times(2, 4)           # Arguments in parentheses
8
```

This expression passes two arguments to `times`. As mentioned previously, arguments are passed by assignment, so in this case the name `x` in the function header is assigned the object 2, `y` is assigned 4, and the function's body is run.

For this function, the body is just a `def`'s return statement or a `lambda`'s expression, that sends back the result as the value of the call expression. The returned object was printed here interactively (as in most languages, $2 * 4$ is 8 in Python), but if we needed to use it later we could instead assign it to a variable. For example:

```
>>> x = times(3.14, 4)    # Save the result object
>>> x
12.56
```

Now, watch what happens when the function is called a third time, with very different kinds of objects passed in:

```
>>> times('Py', 4)       # Functions are "typeless"
'PyPyPyPy'
```

This time, our function means something wholly different. In this third call, a string and an integer are passed to `x` and `y`, instead of two numbers. Recall that `*` works on both numbers and sequences; because we don't constrain the types of variables, arguments, or return values in Python, we can use `times` to either *multiply* numbers or *repeat* sequences.

In other words, what our `times` function means and does depends on what we pass into it. This is a core idea in Python (and perhaps the key to using the language well), which merits a separate callout here.

Polymorphism in Python

As we just saw, the very meaning of the expression `x * y` in our simple `times` function depends completely upon the kinds of objects that `x` and `y` are—thus, the same function can perform multiplication in one instance and repetition in another. Python leaves it up to the *objects* to do something reasonable for the syntax. Really, `*` is just a dispatch mechanism that routes control to the objects being processed.

This sort of type-dependent behavior is known as *polymorphism*, a term we first met in [Chapter 4](#) that essentially means that the meaning of an operation depends on the objects being operated upon. Because it's a dynamically typed language, polymorphism runs rampant in Python. In fact, *every* operation is a polymorphic operation in Python: printing, indexing, the `*` operator, and much more.

This is deliberate, and it accounts for much of the language's conciseness and flexibility. A single function, for instance, can generally be applied to a whole category of object types automatically. As long as those objects support the expected *interface* (a.k.a. protocol), the function can process them. That is, if

the objects passed into a function have the expected methods and expression operators, they are plug-and-play compatible with the function's logic.

Even in our simple `times` function, this means that *any* two objects that support a `*` will work, no matter what they may be, and no matter when they are coded. This function will work on two numbers (performing multiplication), or a string and a number (performing repetition), or any other combination of objects supporting the expected interface—even class-based objects we have not even imagined yet.

Moreover, if the objects passed in do *not* support this expected interface, Python will detect the error when the `*` expression is run and raise an exception automatically. It's therefore usually pointless to code error checking in such code ourselves. In fact, doing so would limit our function's utility, as it would be restricted to work only on objects whose types we test for:

```
>>> times('not', 'quite')
TypeError: can't multiply sequence by non-int of type 'str'
```

This turns out to be a crucial philosophical difference between Python and statically typed languages like C++ and Java: in Python, your code is *not supposed to care* about specific data types. If it does, it will be limited to working on just the types you anticipated when you wrote it, and it will not support other compatible object types coded now or in the future. Although it is possible to test for types with tools like the `type` built-in function, doing so breaks your code's flexibility. By and large, we code to object *interfaces* in Python, not data types.

Of course, some programs have unique requirements, and this polymorphic model of programming means we have to *test* our code to detect some errors that statically typed compiled languages might be able to detect earlier. In exchange for an initial bit of extra testing, though, we radically reduce the amount of code we have to write, and radically increase our code's flexibility. As you'll come to find with time, it's a resounding net win in practice.

A Second Example: Intersecting Sequences

Next, let's explore a second function example that does something a bit more useful than multiplying arguments and further illustrates function basics.

In [Chapter 13](#), we coded a `for` loop that collected items held in common in two strings (and called it nearly intersection, except for duplicates). We noted there that the code wasn't as useful as it could be because it was set up to work only on specific variables and could not be rerun later. Of course, we could copy the code and paste it into each place where it needs to be run, but this solution is neither good nor general—we'd still have to edit each copy to support different sequence names, and changing the algorithm would then require changing multiple copies.

Definition

By now, you can probably guess that the solution to this dilemma is to package the `for` loop inside a function. Doing so offers a number of advantages:

- Putting the code in a function makes it a tool that you can run as many times as you like.
- Because callers can pass in arbitrary arguments, functions are general enough to work on any two sequences (or other iterables) you wish to intersect.

- When the logic is packaged in a function, you have to change code in only *one* place if you ever need to change the way the intersection works.
- Coding the function in a module file means it can be imported and reused by any program run on your machine.

In effect, wrapping the code in a function makes it a general utility, as captured by [Example 16-1](#).

Example 16-1. *inter1.py*

```
def intersect(seq1, seq2):
    res = []                # Start empty
    for x in seq1:          # Scan seq1
        if x in seq2:       # Common item?
            res.append(x)   # Add to end
    return res
```

The transformation from the simple code of [Chapter 13](#) to this function is straightforward; we’ve just nested the original logic under a `def` header and made the objects on which it operates passed-in parameter names. Because this function computes a result, we’ve also added a `return` statement to send a result object back to the caller.

Calls

Before you can call a function, you have to make it. To do this, run its `def` statement, either by typing it interactively or by coding it in a module file and importing the file. You can paste it at a REPL as a whole, but we’ll import it from a file here (per [Chapter 3](#), make sure you can see it in your REPL by working in the file’s folder if needed). Once you’ve run the `def` by such means, you can call the function by passing any two sequence objects in parentheses:

```
>>> from inter1 import intersect    # Get function from module
>>> s1 = 'HACK'
>>> s2 = 'CHOK'
>>> intersect(s1, s2)              # Pass two strings
['H', 'C', 'K']
```

Here, we’ve passed in two strings, and we get back a list containing the characters in common. The algorithm the function uses is simple: “For every item in the first argument, if that item is also in the second argument, append the item to the result.” It’s a little shorter to say that in Python than in English, but it works out the same.

To be fair, our `intersect` function could probably be quicker (it executes naive nested loops), isn’t really mathematical intersection (there may be duplicates in the result), and isn’t required at all (as we’ve seen, Python’s set objects provide a built-in intersection operation kicked off with `&`). Indeed, the function could be replaced with a single list comprehension expression, as it exhibits the classic collector-loop code pattern:

```
>>> [x for x in s1 if x in s2]
['H', 'C', 'K']
```

Which also works as a loop inside a `lambda` body expression—despite being five lines in the `def`. This suffices as a demo, but doesn’t make much sense in a simple role like this (again, we’ll put lambdas to better use later):

```
>>> intersect = lambda seq1, seq2: [x for x in seq1 if x in seq2]
>>> intersect(s1, s2)
['H', 'C', 'K']
```

However it's coded, though, this function does the job: its single piece of code can apply to an entire range of object types, as the next section explains. In fact, we'll improve and extend this to support arbitrarily many operands in [Chapter 18](#), after we uncover more about argument passing modes.

Polymorphism Revisited

Like all good functions in Python, `intersect` is polymorphic. That is, it works on arbitrary types, as long as they support the expected object interface:

```
>>> x = intersect([1, 2, 3], (1, 4))      # Pass mixed types
>>> x                                     # Saved result object
[1]
```

This time, we passed in different types of objects to our function—a list and a tuple (mixed types)—and it still picked out the common items. Because you don't have to specify the types of arguments ahead of time, the `intersect` function happily iterates through any kind of objects you send it, as long as they support the expected interfaces.

For `intersect`, this means that the first argument has to support the `for` loop, and the second has to support the `in` membership test, in both `def` and `lambda` flavors. Any two such objects will work, regardless of their specific types—that includes physically stored sequences like strings and lists; all the iterable objects we met in [Chapter 14](#), including files and dictionaries; and even any class-based objects we code that apply operator overloading techniques we'll discuss later in the book.¹

And here again, if we pass in objects that do *not* support these interfaces (e.g., numbers), Python will automatically detect the mismatch and raise an exception for us—which is exactly what we want, and the best we could do on our own if we coded explicit type tests:

```
>>> intersect([1, 2, 3], 1)
TypeError: argument of type 'int' is not iterable
```

By not coding type tests and allowing Python to detect the mismatches for us, we both reduce the amount of code we need to write, and avoid artificially limiting our code's scope.

Segue: Local Variables

Perhaps the most interesting part of this example, though, is its names. It turns out that the variable `res` inside the `def` version of `intersect` is what in Python is called a *local variable*—a name that is visible only to code inside the function `def` and that exists only while the function call runs. In fact, because all names *assigned* in any way inside a function are classified as local variables by default, nearly all the names in the `def` qualify:

¹ This code will always work if we intersect files' contents obtained with `file.readlines()`. It may not work to intersect lines in open input files directly, though, depending on the file object's implementation of the `in` operator or general iteration. Files must generally be rewound (e.g., with a `file.seek(0)` or another `open`) after they have been read to end-of-file once, and so are single-pass iterators. As you'll see in [Chapter 30](#) when we study operator overloading, objects implement the `in` operator either by providing the specific `__contains__` method or by supporting the general iteration protocol with the `__iter__` or older `__getitem__` methods; classes can code these methods arbitrarily to define what iteration means for their data.

- `res` is obviously assigned, so it is a local variable.
- Arguments are passed by assignment, so `seq1` and `seq2` are, too.
- The `for` loop assigns items to a variable, so the name `x` is also local.

All these local variables appear when the function is called and disappear when the function exits—the `return` statement at the end of `intersect` sends back the result *object*, but the *name* `res` goes away. The same goes for argument names in the `lambda`, though its compression hides its loop variable (sans a nested `:=` to export it).

Because of this, a function’s variables don’t clash with names elsewhere, but also won’t remember values between calls. Although the object returned by a function lives on, retaining other *state information* requires other techniques. To fully explore the notion of locals and state, though, we need to move on to the scopes coverage of the next chapter.

Chapter Summary

This chapter introduced the core ideas behind function definition—the syntax and operation of the `def` and `return` statements and `lambda` expression, the behavior of function call expressions, and the notion and benefits of polymorphism in Python functions. As we saw, a `def` and `lambda` are executable code that create a function object at runtime; when the function is later called, objects are passed into it by assignment (which means object reference in Python), and computed values are sent back by `return` in `def` and implicitly in `lambda`. We also began exploring the concepts of local variables and scopes, but saved the details for [Chapter 17](#). First, though, a quick quiz.

Test Your Knowledge: Quiz

1. What is the point of coding functions?
2. At what time does Python create a function?
3. What does a `def` function return if it has no `return` statement in it?
4. When does the code nested inside the function definition run?
5. What’s wrong with checking the types of objects passed into a function?

Test Your Knowledge: Answers

1. Functions are the most basic way of avoiding code *redundancy* in Python—factoring code into functions means that we have only one copy of an operation’s code to update in the future. Functions are also the basic unit of code *reuse* in Python—wrapping code in functions makes it a reusable tool, callable in a variety of programs. Finally, functions allow us to *divide* a complex system into manageable parts, each of which may be developed individually. Functions are also used for object *methods*, but we’re postponing this role until we study classes.

2. A function is created when Python reaches and runs a `def` statement, which creates a function object and assigns it to the function's name. This normally happens when the enclosing module file is imported by another module (recall that imports run the code in a file from top to bottom, including any `defs`), but it can also occur when a `def` is typed interactively or nested in other statements, such as `ifs`. Functions are also created when a `lambda` is reached and run, though this doesn't normally happen during an import operation.
3. A `def` function returns the `None` object by default if the control flow falls off the end of the function body without running into a `return` statement. Such functions are usually called with expression statements, as assigning their `None` results to variables is generally pointless. A `return` statement with no expression in it also returns `None`. A `lambda` implicitly returns its expression's result, so it has no default.
4. The function *body* (the code nested inside the function `def` statement or following the colon in `lambda`) is run when the function is later invoked with a call expression. The body runs anew each time the function is called.
5. Checking the types of objects passed into a function effectively breaks the function's *flexibility*, constraining the function to work on specific types only. Without such checks, the function would likely be able to process an entire array of object types—any objects that support the interface expected by the function will work. (The term *interface* means the set of methods and expression operators the function's code runs.)

Scopes

The preceding chapter introduced basic function definitions and calls. As we saw, Python's core function model is simple to use, but even simple function examples quickly lead to questions about the meaning of variables in code. This chapter moves on to present the details behind Python's *scopes*—the places where variables are defined and looked up. Like module files, scopes help prevent same-name clashes across a program's code: names defined in one program unit don't interfere with names in another.

As you'll learn here, the place where a name is assigned in your code is crucial to determining what the name means. You'll also find that scope usage can have a major impact on program maintenance effort; overuse of *globals*, for example, is a generally bad idea. On the plus side, you'll also learn that scopes can provide a way to retain *state information* between function calls, and they offer an alternative to classes in some roles.

Python Scopes Basics

Now that we're starting to write our own functions, we need to get more formal about what names mean in Python. When we use a name in a program, Python creates, changes, or looks up the name in what is known as a *namespace*—a place where names live. And when we talk about names in relation to code, namespaces correspond to *scopes*: the location of a name's assignment in your source code determines the scope of the name's visibility in your program.

Just about everything related to names, including scope classification, happens at assignment time in Python. As we've seen, names in Python spring into existence when they are first assigned values, and they must be assigned before they are used. Because names are not declared ahead of time, Python uses the location of the assignment of a name to associate it with (i.e., *bind* it to) a particular namespace and scope. In other words, the place where you assign a name in your source code determines the namespace it will live in, and hence its scope of visibility.

Besides packaging code for reuse, functions add an extra namespace layer to your programs to minimize the potential for collisions among variables of the same name—by default, all names *assigned* inside a function are associated with that function's namespace, and no other. This rule means that:

- Names assigned inside a `def` can be seen only by the code *within* that `def`. You cannot even refer to such names from outside the function.
- Names assigned inside a `def` do not clash with variables outside the `def`, even if the same names are used elsewhere. A name `X` assigned *outside* a given `def` (i.e., in a different `def` or at the top level of a module file) is a completely different variable from a name `X` assigned *inside* that `def`.
- Names assigned in a `lambda` work the same way, though assignment in their expressions can happen only for arguments and `:=` named assignments, and is much less common than in `def`.

In all cases, the scope of a variable (where it can be used) is wholly determined by where it is assigned in your source code, and has nothing to do with which functions call which. In fact, as you'll learn in this chapter, variables may be assigned in three different places, corresponding to three different scopes:

- If a variable is assigned inside a `def` or `lambda`, it is *local* to that function.
- If a variable is assigned in an enclosing `def` or `lambda`, it is *nonlocal* to nested functions.
- If a variable is assigned outside all `defs` and `lambdas`, it is *global* to the entire file.

We call this *lexical scoping* because variable scopes are determined entirely by the locations of the variables in the source code of your program files, not by function calls. Hence, a simple visual inspection is enough to make the call.

For example, in the following module file, the `X = 99` assignment creates a *global* variable named `X` (visible everywhere in this file), but the `X = 88` assignment creates a *local* variable `X` (visible only within the `def` statement). If we refer to `X` inside the function, we'll get the function's `X`, not the global:

```
X = 99                                # Global (module) scope X

def func():
    X = 88                            # Local (function) scope X: a different variable
```

Even though both variables are named `X`, their scopes make them different. The net effect is that function scopes help to avoid name clashes in your programs and help to make functions more self-contained program units—their code need not be concerned with names used elsewhere.

Scopes Overview

Before we started writing functions, none of the code we wrote was nested in a `def`, so the names we used either lived in a module or were built-ins predefined by Python like `open` and `zip`. This includes code typed at the REPL: the interactive prompt is technically a module named `__main__` that prints results and doesn't correspond to a real file; in all other ways, though, it's like the top level of a module file.

Functions, though, provide nested namespaces (scopes) that localize the names they use, such that names inside a function won't clash with those outside it (in a module or another function). Specifically, functions define a *local scope* and modules define a *global scope* with the following properties:

- **The enclosing module is a global scope.** Each module is a global scope—that is, a namespace in which variables created by assignment at the top level of the module file live. Global variables become attributes of a module object to the outside world after imports but can also be used as simple variables within the module file itself.

- **The global scope spans a single file only.** Don't be fooled by the word “global” here—names at the top level of a file are global to code within that single file only. There is no notion of an all-encompassing global scope that corresponds to user code in Python (built-ins are truly global, but not user defined). Instead, names are partitioned into modules, and you must always import a module explicitly if you want to be able to use the names its file defines. When you hear “global” in Python, think “module.”
- **Assigned names are local unless declared global or nonlocal.** By default, all the names assigned inside a function definition are put in its local scope—the namespace associated with the function itself. To assign a name that lives at the top level of the module enclosing a function, declare it in a `global` statement inside the function. To assign a name that lives in an enclosing `def` instead, declare it in a `nonlocal` statement. Because `lambda` bodies are limited to expressions, these two statements are just for `def`.
- **All other names are enclosing function locals, globals, or built-ins.** Names *not* assigned a value in the function definition are assumed to be either locals defined in a physically *enclosing* function, *globals* that live in the enclosing module's namespace, or *built-ins* in the predefined built-ins module that Python provides. Enclosing scopes can arise for any `def` and `lambda` combo, though `def` can't be coded in `lambda`.
- **Each call to a function creates a new local scope.** Every time you call a function, you create a *new* local scope—that is, a namespace in which the names created inside that function will live, barring `global` or `nonlocal` statements. You can think of each `def` statement and `lambda` expression as defining a new local scope, but the local scope actually corresponds to a function *call*, for reasons the next bullet explains.
- **Per-call scopes matter in recursion and closures.** It's crucial that each active call receive its own copy of the function's local variables when using *recursion*, an advanced technique that allows functions to loop by calling themselves, and noted briefly in [Chapter 9](#) for comparisons. Recursion is useful in functions we write as well, to process structures whose shapes can't be predicted ahead of time; we'll explore it more fully in [Chapter 19](#). Per-call scopes are also required for the *closure* functions covered ahead here: each call gets a fresh packet of the function's locals, which can remember state-information objects to be used later.

Beyond those basics, there are three subtleties worth underscoring. First, as noted, code typed at the Python *interactive prompt* lives in a module, too, and follows the normal scope rules: names assigned there are global variables, accessible to the entire interactive session. You'll learn more about modules in the next part of this book.

Second, also bear in mind that *any type of assignment* classifies a name as local or global, based on where the assignment appears. This includes `=` statements and `:=` expressions; module names in `import`; function and argument names in `def`; names in `class` covered later; and so on. If you assign a name in any way within a function, it will be local to that function by default. This includes its arguments listed in its header, but also names in its body's code.

Third, *in-place changes* to objects do not classify names as locals; only actual *name* assignments do. For instance, if the name `L` is assigned to a list at the top level of a module, a statement `L = X` within a function will classify `L` as a local, but `L.append(X)` will not. In the latter case, we are changing the list object that `L` references, not `L` itself—hence, `L` is found in the global scope as usual, and Python happily modifies it without requiring a `global` declaration. As usual, it helps to keep the distinction between names and objects clear: changing an object is not an assignment to a name.

Name Resolution: The LEGB Rule

If the prior section sounds complicated, it really boils down to three simple rules. Within a `def` statement:

- Name *assignments* create or change local names by default.
- Name *references* search at most four scopes: local, then enclosing functions (if any), then global, then built-in.
- Names declared in `global` and `nonlocal` statements map assigned names to enclosing module and function scopes, respectively.

The same goes for `lambda`, except for the last bullet: it doesn't support statements.

In other words, all names assigned inside a function `def` statement or `lambda` expression are locals by default. Functions can freely use names assigned in syntactically enclosing functions and the global scope, but they must declare such nonlocals and globals in order to change them.

Python's name-resolution scheme is usually called the *LEGB rule*, after the names of the scopes it searches:

- When you use an *unqualified* name (not after a `."`) inside a function, Python searches up to four scopes—the local (*L*) scope of the function itself, then the local scopes of any enclosing (*E*) `def`s and `lambdas`, then the global (*G*) scope of the surrounding module, and finally the built-in (*B*) scope—and stops at the first place the name is found. If the name is not found during this search, Python reports an error.
- When you assign a name *inside* a function (instead of just referring to it in an expression), Python always creates or changes the name in the assigner's local scope, unless it's declared to be global or nonlocal there.
- When you assign a name *outside* any function (i.e., at the top level of a module file, or at the interactive prompt), the local and global scope are one and the same—the module's namespace.

Because names must be assigned before they can be used (as we saw in [Chapter 6](#)), there are no automatic components in this model: assignments always determine name scopes unambiguously. [Figure 17-1](#) illustrates Python's four scopes and LEGB rule. Note that the second scope lookup layer, *E*—the scopes of enclosing `def`s or `lambdas`—can technically correspond to more than one lookup level. This layer only comes into play when you nest functions within functions for reasons you'll meet ahead, and is enhanced by the `nonlocal` statement.¹

¹ The scope lookup rule was branded the “LGB rule” in the first edition of this book. The enclosing “E” layer was added later in Python to obviate the task of passing in enclosing scope names explicitly with default arguments—an advanced topic that we'll sample later in this chapter. Since this scope is now addressed by the `nonlocal` statement, the lookup rule might have been better named “LNGB,” but backward compatibility matters in books, too. The present form of this acronym also does not account for the newer obscure scopes of comprehensions and exception handlers, but acronyms longer than four letters tend to defeat their purpose!

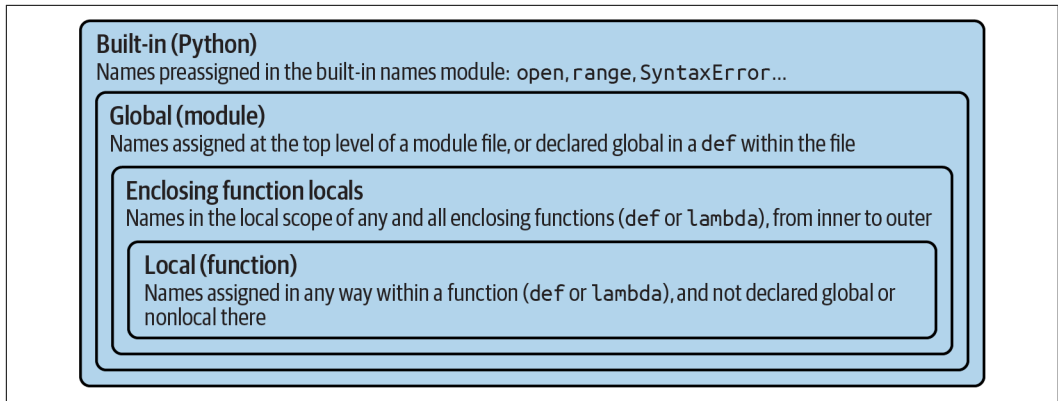


Figure 17-1. The LEGB scope lookup rule for names

Also keep in mind that these rules apply only to simple *variable* names (e.g., `name`). In Parts [V](#) and [VI](#), you'll see that qualified *attribute* names (e.g., `object.name`) live in particular objects and follow a completely different set of lookup rules than those covered here. References to attribute names following periods (`.`) search one or more *objects*, not scopes, and in fact may invoke something called *inheritance* in Python's OOP model; more on this in [Part VI](#).

Preview: Other Python scopes

Though obscure at this point in the book, there are technically three more scopes in Python—temporary loop variables in comprehensions, exception reference variables in try handlers, and local scopes in class statements. The first two of these are special cases that rarely impact real code, and the third falls under the LEGB umbrella rule.

Importantly, most statement blocks and other constructs do not localize the names used within them. This includes loop variables in for loop statements, and the targets of `:=` named assignment, which works the same as all others with regard to scopes. There are, however, some boundary cases whose variables are not available to (and do not clash with) surrounding code, and which involve topics covered in full elsewhere in this book:

- *Comprehension loop variables*: The variable (or variables) `X` in `[X for X in I]` used to refer to the current iteration item in comprehension expressions. Because they might clash with other names and reflect internal state in generators, such names are local to the expression itself in all comprehension forms: list, dictionary, set, and generator. By contrast, for loop *statements* never localize their loop variable (e.g., `X in for X in I:`) to the statement block as noted, despite the similarity. Moreover, names assigned by `:=` within a comprehension *do* leak out to the enclosing scope, per the info and demos in [Chapter 20](#).
- *Exception variables*: The variable `X` in `except E as X` used to reference the raised exception in a try statement handler. Because this might defer garbage collection's memory recovery, such variables are local to that except block, and in fact are *removed* from the containing scope when the block is exited (even if you've used them earlier in your code!). You'll learn all about try statements in [Chapter 34](#).

- *Named assignments in lambda*: The variable *X* in *X := Y* used in assignment expressions. In terms of scopes, the *:=* works the same as an unnested *=* assignment statement in general. Keep in mind, though, that a `lambda` function expression creates a new local function scope just like a `def` statement. Hence, any names assigned by a *:=* nested in a `lambda` are local variables that can be used within the body of the `lambda` itself, but are not available to code outside the `lambda` expression. That said, this is likely a rare coding pattern in most code (and pushes the envelope on complexity). For a refresher on *:=*, see [Chapter 11](#).

Most of these contexts *augment* the LEGB rule, rather than modifying it. Loop variables assigned in a comprehension, for example, are simply bound to a further nested and special-case local scope; all other names referenced within these expressions follow the usual LEGB lookup rules.

It's also worth noting that the `class` statement you'll meet in [Part VI](#) creates a new *local* scope, too, for the names assigned inside the top level of its block. As for `def`, names assigned inside a `class` don't clash with names elsewhere, and follow the LEGB lookup rule, where the `class` block is the *L* level. As for imported modules, these class-local names also morph into class object attributes after the `class` statements ends. For all practical purposes, classes are a local scope (despite some rare border cases that likely qualify as glitches, and can be safely omitted here).

Unlike functions, though, class names are not created per *call*: class object calls generate *instances*, which inherit names assigned in the class and record per-object state as per-instance attributes of their own. As you'll also learn in [Chapter 29](#), although the LEGB rule is used to resolve names used in both the top level of a class itself as well as the top level of method functions nested within it, classes themselves are *skipped* by scope lookups—their names must be fetched as object attributes. Hence, because Python searches enclosing functions for referenced names, but not enclosing classes, the LEGB rule sketched in [Figure 17-1](#) still applies to OOP code.

Scopes Examples

But enough theory! Let's step through a live example that demos scope ideas. Suppose we wrote the code in [Example 17-1](#) and saved it in a module file named *scopes101.py*.

Example 17-1. *scopes101.py*

```
# Global scope
X = 99                # X and func assigned in module: global

def func(Y):          # Y and Z assigned in function: locals
    # Local scope
    Z = X + Y         # X is a global when referenced here
    return Z

func(1)               # func in module: result=100 (not printed)
```

To see this example's result, import and call its function; the file's global `func` is a module attribute to importers (be sure to run this in the same folder as the file if it matters for imports in your REPL, as introduced in [Chapter 3](#)):

```
>>> import scopes101
>>> scopes101.func(1)
100
```


This module and the function it contains use a number of names to do their business. Applying Python's scope rules, we can classify these names as follows:

Global names: X, func

X is global because it's assigned at the top level of the module file; it can be referenced inside the function as a simple unqualified variable without being declared global. func is global for the same reason; the def statement makes a function object and assigns it to the name func at the top level of the module.

Local names: Y, Z

Y and Z are local to the function (and exist only while the function runs) because they are both assigned values in the function definition: Z by virtue of the = statement, and Y because arguments are always passed by assignment. Hence both are assigned during a function call, and both are locals.

The underlying rationale for this name-segregation scheme is that local variables serve as *temporary* names that you need only while a function is running. For instance, in the preceding example, the argument Y and the addition result Z exist only inside the function; these names don't interfere with the enclosing module's namespace—or any other function, for that matter. In fact, local variables are removed from memory when the function call exits, and objects they reference may be *garbage-collected* if not referenced elsewhere. This is an automatic internal step, but it helps minimize memory requirements.

The local/global distinction also makes functions easier to understand, as most of the names a function uses appear in the function itself, not at an arbitrarily distant place in a module file. Also, because you can be sure that local names will not be changed by some remote function in your program, they tend to make programs easier to debug and modify. In the following sort of code, for instance, each function's variable cannot be changed—or even seen—anywhere else:

```
def func1():
    X = 'hack'          # This X...

def func2():
    X = 'code'          # ...is not the same as this X
```

The net effect helps make functions self-contained units of software, by design.

The Built-in Scope

We've been talking about the built-in scope in the abstract, but it's a bit simpler than you may think. Really, the built-in scope is just a built-in module called `builtins`, but you have to import `builtins` to query built-ins because the name `builtins` is not itself a built-in...

Yes, seriously! The built-in scope is implemented as a standard-library module named `builtins`, but that name itself is not placed in the built-in scope, so you have to import it in order to inspect it. Once you do, you can run a `dir` call to see which names are predefined (run this on your own for full fidelity: it can vary across Python versions):

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BaseExceptionGroup', 'BlockingIOError', 'BrokenPipeError', 'BufferError',
...many more names omitted: 158 total in 3.12...
'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed',
```

```
'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'vars', 'zip']
```

The names in this list constitute the built-in scope in Python. Roughly the first half are built-in exceptions, and the second half are built-in functions. Also in this list are the special names `None`, `True`, `False`, and `Ellipsis` we met earlier. Because Python automatically searches this module last in its LEGB lookup (it's *B*), you get all the names in this list “for free.” That is, you can use them without importing any modules. Thus, there are really two ways to refer to a built-in function—by taking advantage of the LEGB rule, or by manually importing the `builtins` module:

```
>>> zip                                     # The normal way, per LEGB's B
<class 'zip'>

>>> import builtins                         # The hard way: for customizations
>>> builtins.zip
<class 'zip'>

>>> zip is builtins.zip                    # Same object, different lookup routes
True
```

The second of these approaches, though more to type, can be useful in advanced roles you'll meet in the sidebar [“Why You Will Care: Customizing open” on page 421](#).

Redefining built-in names: For better or worse

Careful readers might notice that because the LEGB lookup procedure takes the *first* occurrence of a name that it finds, names in the local scope may override variables of the same name in both the global and built-in scopes, and global names may override built-ins. A function can, for instance, create a local variable called `open` by assigning to it:

```
def hider():
    open = 'text'                          # Local variable, hides built-in here (L before B)
    ...
    open('data.txt')                      # Error: this no longer opens files in this scope!
```

However, this will *hide* the built-in function called `open` that lives in the built-in (outer) scope, such that the name `open` will no longer work within the function to open files—it's now a string, not the opener function. This is sometimes called “shadowing” a built-in. This isn't a problem if you don't need to open files in this function, but triggers an error if you attempt to open through this name in this scope; your `open` is no longer the built-in `open`.

This can even occur more simply at the interactive prompt, which works as a global, implicit-module scope:

```
>>> open = 99                             # Assign in global scope, hides built-in here too
```

That being said, there is nothing inherently *wrong* with using a built-in name for variables of your own, as long as you don't need the original built-in version. After all, if these were truly off limits, we would need to memorize the entire built-in names list and treat all its names as reserved. With some 150 usable names in this module in Python 3.12, that would be far too restrictive and daunting:

```
>>> len(dir(builtins)), len([x for x in dir(builtins) if not x.startswith('__')])
(158, 150)
```

In fact, there are times in advanced programming where you may really *want* to replace a built-in name by redefining it in your code—to define a custom `open` that augments access attempts, for instance (again, more in the sidebar at the end of the chapter). Exception: the special names `None`, `True`, and

False in the built-in scope are also treated as reserved words today, so they can no longer be reassigned (fun though it once was!). All other built-in names, though, are fair game.

Nevertheless, redefining a built-in name is often a bug, and a nasty one at that, because Python will not issue a warning message about it. You may find tools on the web that can warn you of such mistakes, but knowledge may be your best defense on this point: don't redefine a built-in name you need. If you *accidentally* reassign a built-in name at the interactive prompt this way, though, you can either restart your session or run a `del name` statement to remove the redefinition from your scope, thereby restoring the original in the built-in scope per LEGB.

Note that functions can similarly hide *global* variables of the same name with locals, but this is more broadly useful, and in fact is much of the point of local scopes—because they minimize the potential for name clashes, your functions are self-contained namespaces:

```
x = 88                                # Global X

def func():
    x = 99                            # Local X: hides global, but we want this here

func()
print(X)                             # Prints 88: unchanged
```

Here, the assignment within the function creates a local `x` that is a completely different variable from the global `x` in the module outside the function. As a consequence, though, there is no way to *change* a name outside a function without adding a `global` or `nonlocal` declaration to the `def`; the next section takes up the former.



More built-in tongue twisters: Technically, the name `__builtins__` is preset in most global scopes, including the interactive session, to reference the module known as `builtins`, so you can often use `__builtins__` without an import, but cannot run an import on the name `__builtins__` itself—it's a preset variable, not a module's name. Thus, `builtins` is `__builtins__` is `True` after you `import builtins`. The upshot is that we can often inspect the built-in scope by simply running `dir(__builtins__)` sans imports, but are advised to use `builtins` for real work and customization. Who said documenting this stuff was easy?

The global Statement

The `global` statement and its `nonlocal` cousin are the only declaration statements in Python that are actually used by Python (for contrast, see [Chapter 6](#)'s discussion of unused type hinting). They are not type or size declarations, though; they are *namespace declarations*. The `global` statement, for instance, tells Python that a function plans to change one or more global names—that is, names that live in the enclosing module's scope (namespace).

We've talked about `global` in passing already. Here's a summary:

- Global names are variables assigned at the top level of the enclosing module file.
- Global names must be declared only if they are assigned within a function.
- Global names may be referenced within a function without being declared, per the LEGB rule.

In other words, `global` allows us to *change* names that live outside a `def` at the top level of a module file. As you’ll see later, the `nonlocal` statement is almost identical but applies to names in an enclosing `def`’s local scope, rather than names in the enclosing module.

The `global` statement, usable in `def` but not `lambda`, simply consists of the reserved word `global`, followed by one or more names separated by commas. All the listed names will be mapped to the enclosing module’s scope when assigned or referenced within the function body. For instance, the following is a takeoff on the preceding example:

```
X = 88                                # Global X

def func():
    global X
    X = 99                            # Global X: outside def

func()
print(X)                             # Prints 99
```

We’ve added a `global` declaration to the example here, such that the `X` inside the `def` now refers to the `X` outside the `def`; they are the same variable this time, so changing `X` inside the function changes the `X` outside it. Here is a slightly more involved example of `global` at work:

```
y, z = 1, 2                          # Global variables in module
def all_global():
    global x                          # Declare globals assigned
    x = y + z                        # No need to declare y or z: LEGB rule
```

Here, `x`, `y`, and `z` are all globals inside the function `all_global`. Names `y` and `z` are global because they aren’t assigned in the function, and `x` is global because it was listed in a `global` statement to map it to the module’s scope explicitly. Without the `global` here, `x` would be considered local by virtue of the assignment.

Notice that `y` and `z` are not declared global; Python’s LEGB lookup rule finds them in the module (`G`) automatically. Also notice that `x` does not even exist in the enclosing module before the function runs; in this case, the first assignment in the function creates `x` in the module. All of which works when needed, but you really should try to avoid using globals like this whenever possible—as the next section will explain.

Program Design: Minimize Global Variables

Functions in general, and global variables in particular, raise some larger design questions. How, for example, should functions communicate? Although some answers will become more apparent when you begin writing larger functions of your own, a few guidelines up front might spare you from problems later. In general, functions should rely on *arguments* and *return* values instead of globals, but this may not at all be obvious to newcomers to programming.

By default, names assigned in functions are locals, so if you want to change names outside functions you have to write *extra* code (e.g., `global` statements). This is deliberate—you have to say more to do the potentially “wrong” thing. Although there are times when globals are useful (and even required), variables assigned in a `def` are local by default because that is normally the best policy. Changing globals can lead to well-known software-engineering problems: because the variables’ values are dependent on the *order* of calls to arbitrarily distant functions, programs can become difficult to debug, or understand at all.

Consider this module file, for example, which is presumably imported and used elsewhere:

```
X = 99

def func1():
    global X           # Change global X when called
    X = 88

def func2():
    global X           # But so does this, and when?
    X = 77
```

Now, imagine that it is your job to modify or reuse this code. What will the value of `X` be here? Really, that question has no meaning unless it's qualified with a point of reference in *time*—the value of `X` is timing-dependent, as it depends on which function was called last (something we can't tell from this file alone).

The net effect is that to understand this code, you have to trace the flow of control through the *entire program*. Hence, if you need to reuse or modify the code, you have to keep the entire program in your head all at once. In fact, you can't really use one of these functions without bringing along the other. They are dependent on—that is, *coupled* with—the global variable. And that is the problem with globals: they generally make code more difficult to understand and reuse than code consisting of self-contained functions that rely on locals.

On the other hand, short of using tools like the nested scope closures covered ahead or OOP with classes covered later, global variables are the most basic way to retain shared *state information*—information that a function needs to remember for use the next time it is called. Local variables disappear when the function returns, but globals do not. As you'll see later, other techniques can achieve this, too, and allow for multiple copies of the retained information; but they are more complex than pushing values out to the global scope for retention in simple cases where this applies.

Moreover, some programs designate a single module to collect shared globals; as long as this is expected, it is not as harmful. Programs that use multithreading for parallel processing also commonly depend on global variables—they become shared memory between functions running in parallel threads, and so act as a communication device.²

For now, though, and especially if you are relatively new to programming, avoid the temptation to use globals whenever you can—they tend to make programs difficult to understand and reuse, and won't work for cases where one copy of saved data is not enough. That's why they are not the default in Python. Try to communicate with passed-in arguments and return values instead. Six months from now, both you and your coworkers may be glad you did.

2 *Multithreading* runs function calls in parallel with the rest of a program and is supported by Python's standard-library modules `_thread`, `threading`, and `queue`. Because all threaded functions run in the same process, global scopes often serve as one form of shared memory between them (threads may share both names in global scopes, as well as objects in a process's memory space). Threading is commonly used for long-running tasks in GUIs, to implement nonblocking IO, and to utilize CPU capacity. Threading is also well beyond this language book's scope (a property it shares with `async` functions, which are nevertheless part of Python syntax today, as you'll learn in [Chapter 20](#)). See Python's library manual for more details.

Program Design: Minimize Cross-File Changes

While we're on the subject of globals, here's another related design note: although we *can* change global variables in another file directly, we usually *shouldn't*. Module files were introduced in [Chapter 3](#) and are covered in depth in the next part of this book, but their basics are simple. To demo their relationship to scopes, consider these two module files:

```
# first.py
X = 99                                     # This code doesn't know about second.py

# second.py
import first
print(first.X)                            # OK: references a name in another file
first.X = 88                             # But changing it can be too subtle and implicit
```

The first defines a variable `X`, which the second prints and then changes by assignment. Notice that we must import the first module into the second to get to its variable at all—as we've learned, each module is a self-contained namespace (package of variables), and we must import one module to see inside it from another. That's the main purpose of modules: by segregating variables on a per-file basis, they avoid name collisions across files, in much the same way that local variables avoid name clashes across functions.

Really, though, in terms of this chapter's topic, the global *scope* of a module file becomes the attribute *namespace* of the module object once it is imported—importers automatically have access to all of the file's global variables, because a file's global scope morphs into an object's attribute namespace when it is imported.

After importing the first module, the second module prints its variable and then assigns it a new value. Referencing the module's variable to print it is fine—this is how modules are linked together into a larger system normally. The problem with the assignment to `first.X`, however, is that it is far too implicit: whoever's charged with maintaining or reusing the first module probably has no clue that some arbitrarily far-removed module on the import chain can change `X` at runtime. In fact, the second module may be in a different folder, and so difficult to notice at all.

Although such cross-file variable changes are always possible in Python, they are usually much more subtle than you will want. Again, this sets up too strong a *coupling* between two components—because the files are both dependent on the value of the variable `X`, it's difficult to understand or reuse one file without the other. Such implicit cross-file dependencies can lead to inflexible code at best, and surprising bugs at worst.

Here again, and generally speaking, don't do that—the better way to communicate across file boundaries is to call functions, passing in arguments and getting back return values. In this specific case, we would probably be better off coding an *accessor function* to manage the change:

```
# first.py
X = 99

def setX(new):
    global X
    X = new                                # Accessors make external changes explicit
                                         # And can manage access in a single place

# second.py
import first
first.setX(88)                           # Call the function instead of changing directly
```

This requires more code and may seem like a trivial change, but it makes a huge difference in terms of readability and maintainability—when a person reading the first module by itself sees a function, that person will know that it is a point of *interface* and will expect the change to the X. In other words, it removes the element of surprise that is rarely a good thing in software projects. Although we cannot prevent cross-file changes from happening (sans obscure hacks that we'll omit here), common sense dictates that they should be minimized unless widely known across the program.

Other Ways to Access Globals

Interestingly, because global-scope variables morph into the attributes of a loaded module object, we can emulate the `global` statement by importing the enclosing module and assigning to its attributes, as in the module file in [Example 17-2](#). Code in this file accesses its enclosing module, first by importing it, and then by indexing the `sys.modules` dictionary that records all loaded modules and is used in advanced roles (there's more on this dictionary in [Part V](#)).

Example 17-2. `thismod.py`

"Change a global three ways"

```
var = 99                                # Global variable == module attribute

def local():
    var = 0                             # Change local var

def glob1():
    global var                           # Declare global (normal)
    var += 1                             # Change global var

def glob2():
    var = 0                             # Change local var
    import thismod                       # Import myself
    thismod.var += 1                     # Change global var

def glob3():
    var = 0                             # Change local var
    import sys                           # Import system table
    thismod = sys.modules['thismod']     # Get module object (or use __name__)
    thismod.var += 1                     # Change global var

def test():
    print(var)
    local(); glob1(); glob2(); glob3()
    print(var)
```

When we import and call this module's `test` to invoke its other functions, this adds 3 to the global variable `var`—only its first function, `local`, does not impact the global:

```
>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102
```

All these global-access techniques work, and illustrate the equivalence of the global scope to module attributes, but they're noticeably more work than using the `global` statement to make your intentions explicit.

As we've seen, `global` allows us to easily change names in a module outside a function. It has a close relative named `nonlocal` that can be used to change names in enclosing functions, too—but to understand how that can be useful, we first need to explore function nesting in general, the topic we turn to next.

Nested Functions and Scopes

So far, this chapter has largely omitted one part of Python's scope rules on purpose, because it's relatively uncommon to encounter it in practice. However, it's time to take a deeper look at layer *E* in the LEGB lookup rule. This layer was added during Python 2.X's reign. It takes the form of the local scopes of any and all enclosing function's local scopes. Enclosing scopes are sometimes also called *statically nested scopes*. Really, the nesting is a *lexical* one—nested scopes correspond to physically and syntactically nested code structures in your program's source code text.

Nested Scopes Overview

With the addition of nested function scopes, variable lookup rules become slightly more complex. Within a function:

- A *name reference* (`X`) looks for the name `X` first in the current local scope (function); then in the local scopes of any lexically enclosing functions in your source code, from inner to outer; then in the current global scope (the module file); and finally in the built-in scope (which we've seen means the built-in module `builtins`). `global` declarations in a `def` make the search begin in the global (module file) scope instead.
- A *name assignment* (e.g., `X = value`) creates or changes the name `X` in the current local scope, by default. If `X` is declared `global` within the function, the assignment creates or changes the name `X` in the enclosing module's scope instead. If, on the other hand, `X` is declared `nonlocal` within a `def` function, the assignment changes the name `X` in the local scope of the closest enclosing function that assigns the same name.

Notice that the `global` declaration still maps variables to the enclosing module. When nested functions are present, variables in enclosing functions may be referenced, but require `nonlocal` declarations to be changed. Also note that, unlike `global`, `nonlocal` does not create a name in an enclosing `def` if it's not assigned there; `nonlocal` makes sense only if the variables it names are also assigned by an enclosing `def`—and is an error to use otherwise.

This section is primarily concerned with the first bullet in the preceding list: nested scope *references*. We'll explore nested *assignments* and `nonlocal` in a moment, but first we need to grok function nesting in general.

Nested Scopes Examples

Let's turn to code to illustrate some of the preceding points. Here is what an enclosing function scope looks like (type this into a script file or at the interactive prompt to run it live):


```

X = 99                                # Global scope name: not used

def f1():
    X = 88                            # Enclosing def local
    def f2():
        print(X + 1)                 # Reference made in nested def
        f2()

f1()                                  # Prints 89: enclosing def local + 1

```

First off, this is legal Python code: the `def` is simply an executable statement, which can appear anywhere any other statement can—including nested in another `def`. Here, the nested `def` runs while a call to the function `f1` is running; it makes a function and assigns it to the name `f2`, a local variable within `f1`'s local scope. In a sense, `f2` is a temporary function that lives only during the execution of (and is visible only to code in) the enclosing `f1`. As such, its name won't clash with any other part of the program—which is one reason to nest it this way.

But notice what happens inside `f2`: when it uses the variable `X`, it refers to the `X` that lives in the *enclosing* `f1` function's local scope. Because functions can access names in all physically enclosing functions, the `X` in `f2` is automatically mapped to the `X` in `f1`, by the LEGB lookup rule's level *E*.

In fact, this enclosing scope lookup works even if the enclosing function's call has already *ended*. For example, the following code defines a function that makes and *returns* another function, and introduces a common nesting role:

```

def f1():
    X = 88
    def f2():
        print(X + 1)                 # Remembers X in enclosing def scope
        return f2                    # Return f2 but don't call it

    action = f1()                    # Make, return function
    action()                          # Call it now: prints 89

```

In this code, the call to `action` is really running the function we named `f2` when `f1` ran. This works naturally because functions are objects in Python like everything else and can be passed back as return values from other functions. More subtly, `f2` “remembers” the enclosing scope's `X` in `f1`—even though `f1` is no longer active when `f2` is run. Though abstract here, this memory behavior turns out to be one of the main reasons to nest functions, and warrants a closer look in the next section.

Closures and Factory Functions

Depending on whom you ask, the preceding example's behavior is sometimes called a *closure* or a *factory* function—the former describing a *functional programming* technique, and the latter denoting a *design pattern*. In code, a factory function creates and returns a function that has a stateful closure, but the two terms are intertwined.

Whatever the label, the function object in question remembers values in enclosing scopes regardless of whether those scopes are still active for a call. In effect, it has attached packets of memory (a.k.a. *state retention*), which are made anew for each copy of the nested function created, and often provide a simple alternative to classes in this role.

Closures (a.k.a. factory functions) are sometimes used by programs that need to generate event handlers on the fly in response to conditions at runtime. For instance, imagine a GUI that must define actions according to user inputs that cannot be anticipated when the GUI is built and vary per action.

In such cases, we need a function that creates and returns another function, with information (i.e., state) that differs per function made.

To demo this live in simplified terms, consider the following function, typed at the interactive prompt:

```
>>> def maker(N):
    def action(X):
        return X ** N
    return action
```

This defines an outer function that simply makes and returns a nested function, without calling it—`maker` makes `action`, but simply returns `action` without running it. If we then call the *outer* function:

```
>>> f = maker(2)
>>> f
<function maker.<locals>.action at 0x101ae7ba0>
```

What we get back is a reference to the *nested* function built—the one created and assigned to `action` when the nested `def` runs. If we now call what we got back from the outer function:

```
>>> f(3)
9
>>> f(4)
16
```

We invoke the nested function—the one called `action` within `maker`. In other words, we’re calling the nested function that `maker` created and passed back.

Perhaps the most unusual part of this, though, is that the nested function *remembers* integer 2, the value of the variable `N` in `maker`, even though `maker` has returned and exited by the time we call `action`. In effect, `N` from the enclosing local scope is retained as state information attached to the generated `action`, which is why we get back its argument squared whenever it is later called.

Just as important, if we now call the outer function again, we get back a *new* nested function with *different* state information attached. In the following, we compute the argument cubed instead of squared when calling the new function, but the original still squares as before:

```
>>> g = maker(3)
>>> g(4)
64
>>> f(4)
16
```

This works because each call to a closure/factory function like this gets its *own* set of state information. In our demo, the function we assign to name `g` remembers 3, and `f` remembers 2, because each has its own state information retained by the variable `N` in `maker`. In a GUI, `N` might be a username, email address, or other per-function state.

This is all literal inside Python. Nested functions have an attached closure storage area for the enclosing scope names they use. It’s available as the `__closure__` attribute of such functions (and fetching `cell_contents` after indexing this tuple yields a state item), but we don’t need to understand Python internals to use closures in our code.

This is also a somewhat advanced technique that you may not see commonly in most code, and may be more popular among programmers with backgrounds in functional programming languages. On the other hand, enclosing scopes are often employed by the `lambda` function-creation expressions introduced in the prior chapter—because `lambda` is an expression, it is almost *always* nested in a `def`. For

example, a `lambda` can serve in place of a `def` in our demo, and also relies on enclosing scope references to retain state—like `N` in the following variation:

```
>>> def maker(N):
    return lambda X: X * N          # lambda functions retain state too

>>> h, i = maker(2), maker(3)      # Make two closure functions

>>> h('Py'), i('Py')              # Run lambdas: ('Py' * 2) and ('Py' * 3)
('PyPy', 'PyPyPy')
```

For a more tangible example of closures at work, see the sidebar “[Why You Will Care: Customizing open](#)” on page 421. It uses similar techniques to store information for later use in an enclosing scope. As you’ll also see after the next section’s closer, closures become more useful when their state becomes changeable with `nonlocal`.

Arbitrary Scope Nesting

Finally, this tour of nested function scopes would be remiss if it didn’t point out that scopes may nest arbitrarily, though only enclosing functions (not classes, described in [Part VI](#)) are searched when names are referenced:

```
>>> def f1():
    x = 99
    def f2():
        def f3():
            print(x)          # Found in f1's local scope!
            f3()
        f2()

>>> f1()
99
```

This does work: Python will search the local scopes of *all* enclosing `defs`, from inner to outer, after the referencing function’s own local scope and before the module’s global scope or built-ins. However, this sort of code is even less likely to crop up in practice. Per coding aphorism, *flat is better than nested*, and this still holds generally true even with nested scope closures in the toolbox. Except in limited contexts, your life (and the lives of your fellow travelers in the software realm) will generally be better if you minimize nesting in function definitions.

The nonlocal Statement

In the prior section, we saw how nested functions can *reference* variables in an enclosing function’s scope, even if that function has already returned. As suggested earlier, we can also *change* such enclosing scope variables, as long as we declare them in `nonlocal` statements. With this statement, nested functions gain both read and write access to names in enclosing functions. This makes nested scope closures more useful, by making state changeable.

The `nonlocal` statement, usable in `def` but not `lambda`, is similar in both form and role to `global`, covered earlier. Like `global`, `nonlocal` declares that a name (or names) will be changed in an enclosing scope. Unlike `global`, though, `nonlocal` applies to a name in an enclosing function’s scope, not to the global module scope outside all functions. Also unlike `global`, `nonlocal` names must exist in an enclosing function’s scope—they are mapped only to enclosing functions and cannot be created by a first assignment in a nested `def` that uses `nonlocal`.

In other words, `nonlocal` both *allows* assignment to names in enclosing function scopes, and *limits* scope lookups for such names to enclosing functions. The net effect is a direct and reliable implementation of changeable state information, for contexts that do not desire or need to use classes or other stateful tools.

nonlocal Basics

The `nonlocal` statement has meaning only inside a function, and is an error to use elsewhere. More specifically, it applies and is usable only when `def` is nested in another `def`: because the body of a `lambda` allows just an *expression*, it supports neither `nonlocal` nor nested `def` statements.

When used within a `def`, the `nonlocal` statement seals the fate of references—much like the `global` statement, `nonlocal` causes searches for the names listed in the statement to begin in the enclosing `def`s' scopes, not in the local scope of the declaring function. That is, `nonlocal` also means “skip my local scope entirely.” In fact, the names listed in a `nonlocal` *must* be assigned in an enclosing `def`, and never refer to names in the global or built-in scopes.

Importantly, the addition of `nonlocal` does not alter name reference scope rules in general; they still work as before, per the LEGB rule described earlier. The `nonlocal` statement simply serves to allow names in enclosing scopes to be changed rather than just referenced. The `global` and `nonlocal` statements, though, tighten up and even restrict the lookup rules within a function for the names that they list:

- `global` makes scope lookup begin in the enclosing module's scope and allows names there to be assigned. Scope lookup continues on to the built-in scope if the name does not exist in the module, but assignments to global names always create or change them in the module's scope.
- `nonlocal` restricts scope lookup to just enclosing `def`s, requires that the names exist there, and allows them to be reassigned. Scope lookup does not continue on to the global or built-in scopes.

nonlocal in Action

Let's move on to examples to make this more concrete. Simple *references* to enclosing `def` scopes work as we've already seen—in the following, `outer` builds and returns the function `inner` to be called later, and the state reference in `inner` maps the local scope of `outer` using the normal scope LEGB lookup rules:

```
>>> def outer(start):
    state = start                # Referencing nonlocals works normally
    def inner(label):
        print(label, state)      # Remembers state in enclosing scope
    return inner

>>> F = outer(0)
>>> F('code')                  # State is the same on every inner run
code 0
>>> F('hack')
hack 0
```

Changing a name in an enclosing `def`'s scope, however, is not allowed by default:

```
>>> def outer(start):
    state = start
    def inner(label):
        state += 1              # Cannot change by default
```

```

        print(label, state)
    return inner

>>> F = outer(0)
>>> F('code')
UnboundLocalError: cannot access local variable 'state' ...

```

Now, if we declare `state` in the outer scope as `nonlocal` within `inner`, we get to both reference and *change* it inside the nested function. Again, this works even though `outer` has returned and exited by the time we call the returned `inner` function through the name `F`:

```

>>> def outer(start):
    state = start                # Each call gets its own state
    def inner(label):
        nonlocal state          # <= Remembers state in enclosing scope
        state += 1              # Allowed to change it if nonlocal
        print(label, state)
    return inner

>>> F = outer(0)
>>> F('code')                  # Increments state on each call
code 1
>>> F('hack')
hack 2

```

As usual with enclosing scope references, we can call the outer factory (closure) function multiple times to get multiple copies of its state in memory. The `state` object in the enclosing scope is essentially attached to the new `inner` function object returned; each call makes a new, distinct `state` object, such that updating one function’s state won’t impact the other. The following continues the prior listing’s interaction:

```

>>> G = outer(24)              # Make a new outer that starts at 24
>>> G('code')
code 25
>>> G('hack')                  # G's state information updated to 26
hack 26

>>> F('more')                  # But F's state is where it left off: at 2
more 3                         # Each call has different state information

```

As you can see, Python’s `nonlocals` are much more functional than “static” function locals typical in some other languages: in a closure function, `nonlocals` are *per-call*, *multiple copy* data.

nonlocal Boundary Cases

Though useful, `nonlocals` come with some subtleties to be aware of. First, unlike the `global` statement, `nonlocal` names really *must* be assigned in an enclosing `def`’s scope—you cannot create them dynamically by assigning them anew in a nested function. The enclosing function’s assignment can appear either before or after the nested function, but you’ll get an error if it’s missing. In fact, `nonlocals` are checked for an enclosing function assignment at function definition time, before either an enclosing or nested function is ever called:

```

>>> def outer(start):
    def inner(label):
        nonlocal state          # Nonlocals must exist in an enclosing def
        state = 0
        print(label, state)
    return inner

```

```
SyntaxError: no binding for nonlocal 'state' found
```

```
>>> def outer(start):
    def inner(label):
        global state          # But globals don't have to exist when declared
        state = 0
        print(label, state)
    return inner

>>> F = outer(0)
>>> F('glob')
glob 0
>>> state                    # Created by the assignment in nested inner
0
```

Second, `nonlocal` restricts the scope lookup to *just* enclosing `defs`; nonlocals are not looked up in the enclosing module's global scope or the built-in scope outside all `defs`, even if they are already there:

```
>>> state = 0
>>> def outer():
    def inner():
        nonlocal state        # Must be located in a def, not the module
        state += 1            # Use global for globals, not nonlocal
    return inner
```

```
SyntaxError: no binding for nonlocal 'chapter' found
```

```
>>> def outer():
    def inner():
        nonlocal ord          # Ditto for built-ins (and immediate calls)
        print(ord)
    inner()
```

```
SyntaxError: no binding for nonlocal 'ord' found
```

These restrictions make sense once you realize that Python would not otherwise generally know in which enclosing scope to create a brand-new name. In the prior listing, should `state` be assigned in `outer`, or the module outside? Because this is ambiguous, Python must resolve nonlocals at function *creation* time, not function *call* time.

Finally, this chapter has been careful to say that nonlocal names must be assigned in *an* enclosing function, not *the* enclosing function. This can matter when function nesting runs deep: a name listed in `nonlocal` can technically appear *anywhere* in the hierarchy of enclosing functions—not just one level up—and the closest appearance is used. Even so, because such code seems unlikely to crop up in practice (and probably qualifies as cruel and unusual punishment), we'll forgo a formal demo here, but see [“Arbitrary Scope Nesting” on page 409](#) for most of the morass.

State-Retention Options

Given the extra complexity of nested functions, you might wonder why they're worth the fuss. Although it's difficult to see in our small examples, state information becomes essential in many programs. While functions can return results, their local variables won't normally retain other values that must live on between calls. Moreover, many applications require such values to differ per context of use.

Broadly speaking, there are multiple ways for Python functions to retain state between calls. These include the global variables and enclosing scope references we've already met, but also class-instance

attributes and function attributes. To close out this chapter, let's review these options to see how they stack up.

Nonlocals: Changeable, Per-Call, LEGB

First off, the following code is a recap from the prior section, repeated here for compare and contrast. As we've seen, its `nonlocal` allows state in an enclosing scope to be saved and modified. Each call to `outer` makes a self-contained *package of changeable information*, whose names and objects do not clash with any other part of the program:

```
>>> def outer(start):
    state = start                # Each call gets its own state
    def inner(label):
        nonlocal state          # Remembers state in enclosing scope
        state += 1              # Allowed to change it if nonlocal
        print(label, state)
    return inner

>>> F = outer(0)                # State to be saved in enclosing scope
>>> F('nonlocal1')              # State visible within closure only
nonlocal1 1
>>> F('nonlocal2')
nonlocal2 2

>>> F.state
AttributeError: 'function' object has no attribute 'state'
```

We need to declare variables `nonlocal` only if they must be changed (other enclosing scope name references are automatically retained as usual per LEGB), and `nonlocal` names are not visible outside the enclosing function.

While this scheme works well, the next three sections present some alternatives. Some of the code in these sections uses tools we haven't covered yet and is intended partially as preview, but we'll keep the examples simple here so that you can compare and contrast along the way.

Globals: Changeable but Shared

One common suggestion for achieving state retention without `nonlocal` is to simply move saved info out to the *global scope* (the enclosing module):

```
>>> def outer(start):
    global state                 # Move it out to the module to change it
    state = start                # global allows changes in module scope
    def inner(label):
        global state
        state += 1
        print(label, state)
    return inner

>>> F = outer(0)
>>> F('global1')                # Each call increments shared global state
global1 1
>>> F('global2')
global2 2
>>> state                       # State accessible as global outside defs
2
```

This works in this case, but it requires `global` declarations in both functions and is prone to name collisions in the global scope (what if “state” is already being used for something else?). A more subtle problem is that it only allows for a *single shared copy* of the state information in the module scope—if we call outer again, we’ll wind up resetting the module’s state variable, such that prior calls will see their state overwritten:

```
>>> G = outer(24)                                # Resets state's single copy in global scope
>>> G('global3')
global3 25

>>> F('global4')                                # But F's counter has been overwritten!
global4 26
```

As shown earlier, when you use `nonlocal` and nested function closures instead of `global`, each call to outer remembers its own unique copy of the state object. For per-call roles, globals don’t fit the bill.

Function Attributes: Changeable, Per-Call, Explicit

As another state-retention option, we can often achieve the same effect as nonlocals with *function attributes*—user-defined names attached to functions explicitly. When you attach user-defined attributes to nested functions generated by factory functions, they can also serve as per-call, multiple copy, and writeable state, just like `nonlocal` scope closures. Such user-defined attribute names won’t clash with names Python creates itself, and as for `nonlocal`, need be used only for state variables that must be *changed*; other scope references are retained and work normally.

Because factory functions make a new function on each call anyhow, this does not require extra objects—the new function’s attributes become per-call state in much the same way as nonlocals, and are similarly associated with the new function in memory. In contrast, function attributes are perhaps less *magical* than scopes, and allow state variables to be accessed *outside* the nested function. With `nonlocal`, state variables can be seen directly only within the nested `def`; with attributes, state access is a simple function-attribute fetch.

Here’s a mutation of our example based on this technique—it replaces a `nonlocal` with an attribute attached to the nested function. This scheme may not seem as intuitive to some at first glance; you must access state explicitly through the function’s name instead of as simple variables, and must initialize *after* the nested `def`. Still, it allows state to be accessed externally, saves a line by eliminating a `nonlocal` declaration, and makes state usage more explicit:

```
>>> def outer(start):
    def inner(label):
        inner.state += 1                # Change object attr, not inner itself
        print(label, inner.state)       # inner is in the enclosing scope
    inner.state = start                 # Initialize state after inner defined
    return inner

>>> F = outer(0)
>>> F('attr1')                         # F is an inner with state attached
attr1 1
>>> F('attr2')
attr2 2

>>> F.state                             # Can access state outside defs too
2
```


Because each call to the outer function produces a new nested function object, this scheme supports multiple-copy, *per-call* changeable data just like nonlocal closures—a usage mode that global variables cannot provide:

```
>>> G = outer(24)                # G has own state, doesn't overwrite F's
>>> G('attr3')
attr3 25
>>> F('attr4')                  # F's state varies
attr4 3

>>> F.state                      # State is accessible and per-call
3
>>> G.state
25
>>> F is G                       # Different function objects
False
```

Fine points: this code relies on the fact that the function name `inner` is a local variable in the outer scope enclosing `inner`; as such, it can be referenced freely inside `inner` per LEGB's *E*. This code also relies on the fact that changing an object in place is not an assignment to a name; when it increments `inner.state`, it is changing part of the *object* `inner` references, not the name `inner` itself (much like the `L.append` call we saw earlier). Because we're not really assigning a *name* in the enclosing scope, no nonlocal declaration is required.

We'll explore function attributes further in [Chapter 19](#). Importantly, you'll see there that Python uses naming conventions that ensure that the arbitrary names you assign as function attributes won't clash with names related to internal implementation, making the namespace equivalent to a user scope. At the end of the day, function attributes both predate `nonlocal` and provide similar utility, making the latter technically redundant in some roles.



State with enclosing scope mutables: On a related note, it's also possible to change a mutable object like a *list* in the enclosing scope without declaring its name `nonlocal`. The following, for example, implements changeable per-call state information, and largely works the same as the preceding version (though it does not support access to state info from outside the functions):

```
def outer(start):
    def inner(label):
        state[0] += 1          # Clever hack or dark magic?
        print(label, state[0]) # Leverage in-place mutable change
    state = [start]
    return inner
```

This exploits the mutability of lists, and like function attributes, relies on the fact that in-place object changes do not classify a name as local. This is perhaps more obscure than either function attributes or `nonlocal`, though—it's a technique that predates others, and seems to lie today somewhere on the spectrum from arcane hack to dated workaround. You're probably better off using named function attributes than lists and numeric offsets this way (but you can't control what others code).

Classes: Changeable, Per-Call, OOP

Another standard prescription for changeable state information in Python is to use *classes with attributes*. Like function attributes, this scheme makes state information access more explicit than the implicit magic of scope lookup rules. In addition, each instance of a class gets a fresh copy of the state information, as a natural byproduct of Python's object model. Classes also support inheritance, multiple behaviors, and other OOP tools above and beyond functions.

As an abstract and partial illustration, the following defines two stateful instances of a class:

```
class Book:
    ...code to define method functions that manage and use state...

lp6e = Book()          # Make an instance of the class
lp6e.year = 2024       # Access instance state via attributes
lp6e.python = 3.12

lp5e = Book()          # Make another instance of the class
lp5e.year = 2013       # Each instance has its own attributes/state
lp5e.python = 3.3
```

Because classes support a broader array of tools, they tend to require more code than closure functions, but may be better in more demanding roles. We haven't explored classes in any sort of detail yet, though, so we'll have to cut this section short here. Watch for classes, and their explicit flavor of multiple-copy state information, in [Part VI](#).

And the Winner Is...

As a wrap-up, globals, nonlocals, function attributes, and classes all offer changeable state-retention options. Globals support only single-copy shared data, but nonlocals, function attributes, and classes all support multiple-copy changeable state. Of the latter, nonlocals rely on an implicit LEGB lookup, function attributes are manual but explicit and allow state access outside of functions, and classes are a larger solution that comes with OOP's complexities.

As usual, the best tool for your program depends upon your program's goals. We've seen that `nonlocal` provides changeable state for nested functions with a dedicated statement, and is especially useful for simpler state-retention needs where global variables do not apply and classes may not be warranted. That being said, function attributes can often serve the same roles as `nonlocal`, with arguably less implicit behavior.

We'll revisit state-retention options introduced here in [Chapter 39](#) for a more realistic use case—*decorators*, a tool that by nature involves multilevel state retention. State options have additional selection factors (e.g., performance), which we'll have to leave unexplored here for space (you'll learn how to time code speed in [Chapter 21](#)). For now, it's time to explore one more technique, which both confounds the LEGB rule and segues to the next chapter.

Scopes and Argument Defaults

In early versions of Python, the enclosing scope references we've used in this chapter failed because nested functions did not do anything about scopes—a reference to an enclosing function's variable would search only the local, global, and built-in scopes. Because it skipped the scopes of enclosing functions, an error would result. To work around this, programmers typically used *default argument values* to pass in and remember the objects in an enclosing scope.

Though a preview of the next chapter’s arguments coverage, this also bears on scopes. In the following, a takeoff of an earlier example, a value from the enclosing function’s scope is passed into a nested function via an unused argument’s default:

```
def f1():
    X = 88
    def f2(X=X):          # Remember enclosing scope X with defaults
        print(X + 1)
    f2()

f1()                      # Prints 89
```

This syntax also works in `lambda`, which, as we’ve learned, naturally and normally creates nested-function scopes:

```
def f1():
    X = 88
    f2 = lambda X=X: print(X + 1)
    f2()
```

Both of these examples still work in all Python releases, and rely on argument defaults. In short, the syntax *arg=val* in a function header means that the argument *arg* will default to the value *val* if no real value is passed to *arg* in a call. In the preceding code, this is used to explicitly assign enclosing scope state to be retained.

Specifically, in the modified *f2*s here, the *X=X* means that the argument *X* on the left will default to the value of *X* in the enclosing scope—because the *X* on the right is evaluated *before* Python steps into the nested function, it still refers to the *X* in *f1*. In effect, each *f2*’s default argument *X* remembers what *X* was in the enclosing *f1*: the object 88.

That’s fairly subtle, and it depends entirely on the *timing* of default-value evaluations. It’s also an *incidental*—if not accidental—feature: this works only if a real value is never passed to argument *X* to overwrite the default. In fact, the nested scope lookup rule was added to Python in part to make defaults unnecessary for this role: today, Python *automatically* remembers any values required from the enclosing scope for use in nested *defs* and *lambdas*. As we’ve seen, this example today works the same sans defaults:

```
def f1():
    X = 88
    def f2():          # Remember enclosing scope X per LEGB rule
        print(X + 1)   # And likewise for lambda
    f2()
```

That said, flat is generally better than nested again, and function nesting in some such code makes programs more complex than they need be. The following, for instance, is an equivalent of the prior examples that avoids nesting altogether. Notice the forward reference to *f2* inside *f1* in this code—it’s OK to call a function defined *after* the function that calls it, as long as the second *def* runs before the first function is actually *called*. Code inside a function’s body is never evaluated until the function is later called:

```
def f1():
    X = 88              # Pass x along instead of nesting
    f2(X)              # Forward reference OK

def f2(X):
    print(X + 1)       # Flat is still often better than nested
```

If you avoid nesting this way, you can almost forget about the nested scopes concept in Python. On the other hand, nested functions can avoid *name clashes* by localizing the names of functions used nowhere else, and nesting is the basis of *closure functions*, which support stateful callable objects useful in a variety of roles. When functions are nested for such reasons, the LEGB rule almost makes defaults unnecessary for saving state from an enclosing function's scope—*except* in the following case.

Loops Require Defaults, Not Scopes

So why bother learning an outdated scope-reference scheme? Because it's still *required* in one common case: if a `lambda` or `def` is nested in a *loop*, and the nested function references an enclosing scope variable that is changed by that loop, then all functions generated within the loop will have the same value—the value the referenced variable had in the *last* loop iteration. In such cases, you must still use defaults to save the variable's *current* value instead.

This may seem obscure, but it can come up in practice more often than you may think, especially in code that generates event-handler functions for a number of widgets in a GUI—for instance, handlers for button-clicks for all the buttons in a panel. If these are created in a loop (and they often will be), you need to be careful to save state with defaults, or all your buttons' callbacks may wind up doing the same thing.

Here's an illustration of this phenomenon reduced to simple code: the following attempts to build up a list of functions that each remember the current variable `i` from the enclosing scope:

```
>>> def makeActions():
    acts = []
    for i in range(5):
        acts.append(lambda x: i ** x)      # Try to remember each i in 0..4
    return acts                             # But all remember same last i!

>>> acts = makeActions()
>>> acts[0]
<function makeActions.<locals>.<lambda> at 0x101ae7ec0>
```

Interestingly, this can also be coded as a list comprehension, where a `lambda` can be used as the collection result, and the list comprehension serves as a local scope for its loop variable:

```
>>> acts = [(lambda x: i ** x) for i in range(5)]
>>> acts[0]
<function <lambda> at 0x10de6bce0>
```

Either way, though, this doesn't quite work—because the enclosing scope variable `i` is looked up when the nested functions are later *called*, they all effectively remember the same value: the value the loop variable had on the *last* loop iteration. That is, when we pass a power argument of 2 to `x` in each of the following calls, we get back 4 to the power of 2 for each function in the list, because `i` is the same in all of them—4:

```
>>> acts[0](2)          # All are 4 ** 2, 4=last i
16
>>> acts[1](2)          # This should be 1 ** 2 (1)
16
>>> acts[2](2)          # This should be 2 ** 2 (4)
16
>>> acts[4](2)          # Only this should be 4 ** 2 (16)
16
```

In this case, we still have to explicitly retain enclosing scope values with default arguments, rather than enclosing scope references. That is, to make this sort of code work, we must pass in the *current* value of

the enclosing scope's variable with a default. Here's the required mod for both the `def` and comprehension versions:

```
>>> def makeActions():
    acts = []
    for i in range(5):
        acts.append(lambda x, i=i: i ** x) # Use defaults instead
    return acts

>>> acts = [(lambda x, i=i: i ** x) for i in range(5)]
```

In either coding, because defaults are evaluated when the nested function is *created* (not when it's later *called*), each remembers its own value for `i`:

```
>>> acts = makeActions()
>>> acts[0](2)           # 0 ** 2
0
>>> acts[1](2)           # 1 ** 2
1
>>> acts[2](2)           # 2 ** 2
4
>>> acts[4](2)           # 4 ** 2
16
```

Nor are function attributes a fix here: because a function's own *name* is a variable from the enclosing scope that changes in the loop too, in every function it may reference the *last* function made by the *last* loop iteration. We'll omit the gory details here, but keep in mind that *any* enclosing scope reference changed by a loop may require defaults.

This may seem an odd special case, but it reflects Python's implementation of variable scopes, and will become more likely to crop up as you start writing larger programs. This case is also rooted in both scopes and arguments defaults, and to understand the latter in full, we have to move on to the next chapter.



State with argument-default mutables: Also on a related note, it's possible to retain state too with mutable argument defaults like lists and dictionaries (e.g., `def f(a=[])`). Because defaults are implemented as objects attached to functions at function *creation* time, mutable defaults retain state from call to call, rather than being initialized anew on each call. Defaults can also retain mutables from an enclosing function's scope, thereby enabling changeable per-call state information.

Depending on whom you ask—and when you ask them—this is either a feature that supports state retention, or a perilous and dark corner of the language to be avoided. Usually, programmers expect defaults initialized with literals like `[]` to be re-created on every call, and are surprised when they retain prior calls' values. More on this in [“Function Gotchas” on page 549](#).

Chapter Summary

In this chapter, we studied one of two key concepts related to functions: *scopes*, which determine how variables are looked up when used. As we learned, variables are considered local to the function definitions in which they are assigned, unless they are specifically declared to be global or nonlocal. We also explored some more advanced scope concepts here, including nested function scopes and function

attributes. Finally, we looked at some general design ideas, such as the need to minimize globals and cross-file changes.

In the next chapter, we're going to continue our function tour with the second key function-related concept: argument passing. As you'll find, arguments are passed into a function by assignment, but Python also provides tools that allow functions to be flexible in how items are passed, including the defaults we previewed here. Before we move on, let's take this chapter's quiz to review the scope concepts we've covered here.

Test Your Knowledge: Quiz

1. What is the output of the following code, and why?

```
>>> X = 'Hack'
>>> def func():
>>>     print(X)

>>> func()
```

2. What is the output of this code, and why?

```
>>> X = 'Hack'
>>> def func():
>>>     X = 'Py!'

>>> func()
>>> print(X)
```

3. What does this code print, and why?

```
>>> X = 'Hack'
>>> def func():
>>>     X = 'Py!'
>>>     print(X)

>>> func()
>>> print(X)
```

4. What output does this code produce? Why?

```
>>> X = 'Hack'
>>> def func():
>>>     global X
>>>     X = 'Py!'

>>> func()
>>> print(X)
```

5. What about this code—what's the output, and why?

```
>>> X = 'Hack'
>>> def func():
>>>     X = 'Py!'
>>>     def nested():
>>>         print(X)
>>>     nested()

>>> func()
>>> print(X)
```

6. How about this example: what is its output, and why?

```
>>> def func():
    x = 'Py!'
    def nested():
        nonlocal x
        x = 'Hack'
    nested()
    print(x)

>>> func()
```

7. Name three or more ways to retain state information across calls in a Python function.

Test Your Knowledge: Answers

1. The output here is Hack because the function references a global variable in the enclosing module (because it is not assigned in the function, it is considered global).
2. The output here is Hack again because assigning the variable inside the function makes it a local and effectively hides the global of the same name. The `print` statement finds the variable unchanged in the global (module) scope.
3. It prints Py! on one line and Hack on another, because the reference to the variable within the function finds the assigned local and the reference in the `print` statement finds the global.
4. This time it just prints Py! because the global declaration forces the variable assigned inside the function to refer to the variable in the enclosing global scope, even though the variable is assigned inside the function.
5. The output in this case is again Py! on one line and Hack on another, because the `print` statement in the nested function finds the name in the enclosing function's local scope, and the display at the end finds the variable in the global scope.
6. This example prints Hack because the `nonlocal` statement means that the assignment to `x` inside the nested function changes `x` in the enclosing function's local scope. Without this statement, this assignment would classify `x` as local to the nested function, making it a different variable; the code would then print Py! instead.
7. Although the values of local variables go away when a function returns, you can make a Python function retain state information by using shared *global* variables, *nonlocal* enclosing scope references within nested functions, or using *default* argument values. Function *attributes* also allow state to be attached to the function itself, instead of looked up in scopes. Another alternative, using *classes* and OOP, sometimes supports state retention better than any of the scope-based techniques because it makes it explicit with attribute assignments; we'll explore this option in [Part VI](#). Changing *mutable* objects in scopes and defaults works too, but may not be legal in some locales.

Why You Will Care: Customizing open

For another example of closures at work, consider changing the built-in `open` call to a custom version as suggested earlier in this chapter. If the custom version needs to call the original, it must save it before changing it, and retain it for later use—a classic state retention scenario. Moreover, if we

wish to support multiple customizations to the same function, globals won't do: we need per-customizer state.

The following, coded in file *makeopen.py*, is one way to achieve this. It uses a nested scope closure to remember a value for later use, without relying on global variables—which can clash and allow just one value, and without using a class—that may require more code than is warranted here:

```
import builtins

def makeopen(id):
    original = builtins.open
    def custom(*pargs, **kargs):
        print(f'Custom open call {id}', pargs, kargs)
        return original(*pargs, **kargs)
    builtins.open = custom
```

To change `open` for every module in a process, this code reassigns it in the built-in scope to a custom version coded with a nested `def`, after saving the original in the enclosing scope so the customization can call it later. This code is partially a preview, as it relies on *starred-argument* forms to collect and later unpack arbitrary positional and keyword arguments meant for `open`—a topic coming up in the next chapter. Much of the magic here, though, is scope closures: the custom `open` found by the LEGB rule retains the original:

```
>>> file = '../Chapter14/data.txt'
>>> F = open(file)                # Call built-in open in builtins
>>> F.read()
'Testing file IO\nLearning Python, 6E\nPython 3.12\n'

>>> from makeopen import makeopen  # Import open resetter function
>>> makeopen('MOD1')               # Custom open calls built-in open

>>> F = open(file)                # Call custom open in builtins
Custom open call MOD1 ('../Chapter14/data.txt',) {}
>>> F.read()
'Testing file IO\nLearning Python, 6E\nPython 3.12\n'
```

Because each customization remembers the former built-in scope version in its own enclosing scope, they can even be *nested* naturally in ways that global variables cannot support—each call to the `makeopen` closure function remembers its own versions of `id` and `original`, so multiple customizations may be run:

```
>>> makeopen('MOD2')              # Nested customizers work too
>>> F = open(file)                # Because each retains its own state
Custom open call MOD2 ('../Chapter14/data.txt',) {}
Custom open call MOD1 ('../Chapter14/data.txt',) {}
>>> F.read()
'Testing file IO\nLearning Python, 6E\nPython 3.12\n'
```

As is, our function simply adds possibly nested call tracing to a built-in function, but the general technique may have other applications. A class-based equivalent to this may require more code because it would need to save the `id` and `original` values explicitly in object attributes—but requires more background knowledge than we yet have; stay tuned for state retention in classes in this book's [Part VI](#).

Arguments

The preceding chapter explored Python’s *scopes*—the places where variables are defined and looked up. As we saw, the place where a name is defined in our code determines much of its meaning. This chapter continues the function story by studying the concepts in Python *argument passing*—the way that objects are sent to functions as inputs. As you’ll see, arguments (a.k.a. parameters) are assigned to names in a function, but have more to do with object references than with variable scopes. You’ll also find that Python provides extra tools, such as keywords, defaults, and argument collectors and extractors, that allow arguments to be sent to functions flexibly.

Argument-Passing Basics

Earlier in this part of the book, we learned that `def` and `lambda` are function *definitions*, and both include argument-list *headers* that name variables which receive values passed by *calls*. These arguments are used in function bodies, and may be matched between call and header by position, name, and other means we’ll explore later in this chapter.

More fundamentally, though, it was also noted that all Python arguments are passed by *assignment*—which means *object reference*. This has some subtle ramifications that aren’t always obvious to newcomers. Let’s start our arguments adventure by exploring how this works. Here is a rundown of the key points in this model:

- **Arguments are passed by automatically assigning objects to local variable names.** Function arguments are just another instance of Python assignment at work: they are *references* to objects sent by, and possibly shared with, the caller. Because references are implemented as pointers, all arguments are passed by opaque pointer. As for all assignments, objects passed as arguments are never automatically copied.
- **Assigning to argument names inside a function does not affect the caller.** Per assignment norms, when the function is run by a call, argument names in the function header simply become new names in the *local* scope of the function. There is no aliasing between function argument names and variable names in the caller’s scope.

- **Changing a mutable object argument in a function may impact the caller.** On the other hand, as arguments are simply references to passed-in objects, functions can change passed-in mutable objects in place, and the results may affect the caller. Hence, *mutable* arguments can be both input and output for functions.

For more details on *references*, see [Chapter 6](#); everything we studied there also applies to function arguments, though the assignment to argument names is automatic and implicit.

Python's pass-by-assignment scheme isn't quite the same as C++'s reference parameters option, but it turns out to be similar to the argument-passing model of the C language (and others) in practice. You don't need to know these languages to use Python, of course, but the comparison might help those with backgrounds in other tools:

- **Immutable arguments have the same effect as passing “by value.”** Objects such as integers and strings are passed by object reference instead of by copying, but because you can never *change* immutable objects in place anyhow, the net result is much like making a copy in other languages: the caller's values never morph.
- **Mutable arguments have the same effect as passing “by pointer.”** Objects such as lists and dictionaries are also passed by object reference, which has similar consequences to passing arrays as pointers in C: mutable objects can be changed *in place* within the function, with side effects like those for C's arrays.

If you've never used C, Python's argument-passing mode will seem simpler still—it involves just the assignment of object references to names, and it works the same whether the objects are mutable or not.

Arguments and Shared References

To demo argument-passing properties at work, consider the following code:

```
>>> def f(a):                # a is assigned a reference to the passed object
    a = 99                  # Changes local variable a only

>>> b = 88
>>> f(b)                    # a and b both reference same 88 initially
>>> b                        # But b is not changed by assignment to a in f
88
```

In this example, the variable *a* is assigned the object 88 at the moment the function is called with *f(b)*, but *a* lives only within the called function's local scope. Changing *a* inside the function has no effect on the place where the function is called; it simply resets the local variable *a* to a completely different object, 99.

That's what is meant by a lack of name *aliasing*—assignment to an argument name inside a function (e.g., *a=99*) does not magically change a variable like *b* in the scope of the function call. Argument names may share passed *objects* initially (they are essentially pointers to those objects), but only temporarily, when the function is first called. As soon as an argument name is reassigned, this relationship ends.

At least, that's the case for assignment to argument *names* themselves. When arguments are passed *mutable* objects like lists and dictionaries, we also need to be aware that in-place changes to such *objects* may live on after a function exits, and hence impact callers. Here's an example that demonstrates this behavior:

```
>>> def changer(a, b):      # Arguments assigned references to objects
    a = 2                  # Changes local name's value only
    b[0] = 'mod'           # Changes shared object in place: outlives call

>>> X = 1
>>> L = [1, 2]             # Caller:
>>> changer(X, L)          # Pass immutable and mutable objects
>>> X, L                   # X is unchanged, but L is different!
(1, ['mod', 2])
```

In this code, the `changer` function assigns values to argument `a` itself, and to a component of the *object* referenced by argument `b`. These two assignments within the function are only slightly different in syntax but have radically different results:

- Because `a` is a local variable name in the function's scope, the first assignment has no effect on the caller—it simply changes the local variable `a` to reference a completely different object, and does not change the value of the name `X` in the caller's scope. This is the same as in the prior example.
- Argument `b` is a local variable name, too, but it is passed a mutable object—the list that `L` also references in the caller's scope. Because the assignment to `b[0]` in the function is an in-place change to a shared object, its result impacts the value of `L` after the function returns.

Really, the second assignment statement in `changer` doesn't change `b`—it changes part of the object that `b` currently references. This in-place change impacts the caller only because the changed object outlives the function call. The name `L` hasn't changed either—it still references the same, changed object—but it seems as though `L` differs after the call because the value it references has been modified within the function. In effect, the list name `L` serves as both *input* to and *output* from the function.

Figure 18-1 illustrates the name/object bindings that exist immediately after the function has been called, and before its code has run. When the call begins, two objects are shared among four names.

If this example is still confusing, it may help to notice that the effect of the automatic assignments of the passed-in arguments is the same as running a series of simple assignment statements. In terms of the *first* argument, the assignment has no effect on the caller:

```
>>> X = 1
>>> a = X                  # They share the same object
>>> a = 2                  # Name change resets 'a' only, 'X' is still 1
>>> X
1
```

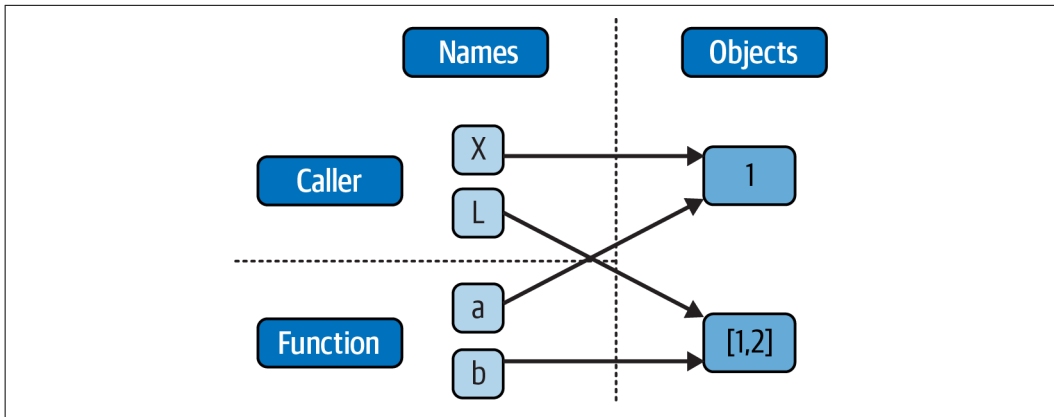


Figure 18-1. Function arguments and shared object references

The assignment through the *second* argument does affect a variable at the call, though, because it is an in-place object change:

```
>>> L = [1, 2]
>>> b = L           # They share the same object
>>> b[0] = 'mod'     # In-place change from 'b': 'L' sees the change too
>>> L
['mod', 2]
```

If you recall our discussions about shared mutable objects in Chapters 6 and 9, you’ll recognize the phenomenon at work: changing a mutable object in place can impact other references to that object. Here, the effect is to make one of the arguments work like both an input and an output of the function.

Avoiding Mutable Argument Changes

This behavior of in-place changes to mutable arguments isn’t a bug—it’s simply the way argument passing works in Python, and turns out to be widely useful in practice. Arguments are normally passed to functions by reference because that is what we normally want. It means we can pass large objects around our programs without making multiple copies along the way, and we can easily update these objects as we go. In fact, as you’ll see in Part VI, Python’s class and OOP model *depends* upon changing a passed-in “self” argument in place, to update mutable object state.

If we don’t want in-place changes within functions to impact objects we pass to them, though, we can simply make explicit copies of mutable objects, as we saw in Chapter 6. For function arguments, we can always copy the list at the point of *call* with tools like `list`, `list.copy`, or an empty slice (dictionaries have similar copy tools):

```
L = [1, 2]
changer(X, L[:])      # Pass a copy, so our 'L' does not change
```

We can also copy within the *function* itself, if we never want to change passed-in objects, regardless of how the function is called:

```
def changer(a, b):
    b = b.copy()       # Copy input list so we don't impact caller
    a = 2
    b[0] = 'mod'        # Changes our list copy only
```

Both of these copying schemes don't stop the function from changing the object—they just prevent those changes from impacting the caller. To really prevent changes, we can always convert to *immutable* objects to force the issue. Tuples, for example, raise an exception when changes are attempted:

```
L = [1, 2]
changer(X, tuple(L))    # Pass a tuple, so changes are errors
TypeError: 'tuple' object does not support item assignment
```

This scheme uses the built-in `tuple` function, which builds a new tuple out of all the items in a sequence (really, any iterable). It's also something of an extreme—because it forces the function to be written to never change passed-in arguments, this solution might impose more limitations on the function than it should, and so should generally be avoided (you never know when changing arguments might come in handy for other calls in the future). Using this technique will also make the function lose the ability to call any list-specific methods on the argument, including methods that do not change the object in place (e.g. `copy`, though tuples have adopted `count` and `index`).

The main point to remember here is that functions might update mutable objects like lists and dictionaries passed into them. This isn't necessarily a problem if it's expected, and often serves useful purposes. Moreover, functions that change passed-in mutable objects in place are probably designed and *intended* to do so—the change is likely part of a well-defined API that you shouldn't violate by making copies.

However, you do have to be aware of this property—if objects change out from under you unexpectedly, check whether a called function might be responsible, and make copies when objects are passed if needed.

Simulating Output Parameters and Multiple Results

Here's another function topic from the assignments department. We've already discussed the `return` statement and used it in examples. What we haven't yet seen is a common though unusual coding technique it enables: because `return` can send back any sort of object, it can return *multiple values* by packaging them in a tuple or other collection type. In fact, although Python doesn't support what some languages label “call by reference” argument passing, we can simulate it by returning tuples and assigning the results back to the original argument names in the caller:

```
>>> def multiple(x, y):
    x = 2                # Changes local names only
    y = [3, 4]
    return x, y          # Return multiple new values in a tuple

>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L) # Assign results to caller's names
>>> X, L
(2, [3, 4])
```

It looks like the code is returning two values here, but it's really just one—a two-item *tuple* with the optional surrounding parentheses omitted. After the call returns, we can use tuple (a.k.a. sequence) assignment to unpack the parts of the returned tuple. (If you've forgotten why this works, flip back to “Tuples” in Chapters 4 and 9, and “Assignments” in Chapter 11.) The net effect of this coding pattern is to both send back multiple results and simulate the *output parameters* of other languages by explicit assignments. Here, `X` and `L` change after the call—but only because the code said so. Lists would work, too, but tuples sans `()` are less to type, and hence common.

Special Argument-Matching Modes

As we’ve just seen, arguments are always passed by *assignment* in Python; names in a `def` or `lambda` definition’s header are assigned references to passed-in objects. On top of this model, though, Python provides additional tools that alter the way the argument objects in a call are *matched* with argument names in the definition prior to assignment. These tools are all optional, but allow us to code functions that support more flexible calling patterns and are commonly used by libraries you’re likely to encounter.

By default, arguments are matched between call and definition by *position*, from left to right, and you must pass exactly as many arguments as there are argument names in the function definition. However, you can also specify matching by name, provide default values, unpack and collect arbitrarily many arguments, and even specify passing-mode requirements. This section presents all these extra tools with a quick overview followed by examples, and a formal look at how they interact at the end after we’ve had a chance to see the basics.

Argument Matching Overview

Before we get into syntax details, it’s important to stress that these special modes are optional and deal only with matching objects to names; the underlying passing mechanism after the matching takes place is still assignment. In fact, some of these tools are intended more for people writing libraries than for application developers. That said, you may stumble across these modes even if you don’t code them yourself, so the following summarizes all the options:

Positionals: matched from left to right

The normal case, which we’ve mostly been using so far, is to match argument values (passed in a call) to argument names (listed in a function definition) by position, from left to right.

Keywords: matched by argument name

Alternatively, callers can explicitly specify which argument in the function is to receive a value, by giving the definition’s name for the argument with `name=value` syntax.

Defaults: specify values for optional arguments that aren’t passed

Functions themselves can specify default values for arguments to receive if the call passes too few values, again using the `name=value` syntax.

Starred collectors: collect arbitrarily many positional or keyword arguments

Function definitions can use special arguments preceded with one or two `*` characters to collect an arbitrary number of arguments after other matching. This feature is sometimes referred to as *varargs*, after a variable-length argument list tool in the C language; in Python, the arguments are collected in a normal object.

Starred unpackers: pass arbitrarily many positional or keyword arguments

Function calls can also use the one or two `*` syntax to unpack argument collections into separate arguments. This is the inverse of a `*` in a function definition—in the definition it means collect arbitrarily many arguments, while in the call it means unpack arbitrarily many arguments, and pass them individually as discrete values.

Keyword-only arguments: arguments that must be passed by name

Function definitions can also use a `*` in their headers to specify arguments that must be passed by name as keyword arguments, not position. This is often used for configuration options that augment primary arguments.

Positional-only arguments: arguments that must be passed by position

As of Python 3.8, function definitions may additionally use a `/` in their header's arguments list to specify that all arguments preceding it must be passed by position, not keyword-argument name.

Argument Matching Syntax

As a reference and preview, [Table 18-1](#) summarizes the syntax that invokes the argument-matching modes.

Table 18-1. Function argument-matching forms

Syntax	Location	Interpretation
<code>func(value)</code>	Caller	Normal argument: matched by position
<code>func(name=value)</code>	Caller	Keyword argument: matched by name
<code>func(*iterable)</code>	Caller	Pass all objects in <i>iterable</i> as individual positional arguments
<code>func(**dict)</code>	Caller	Pass all key/value pairs in <i>dict</i> as individual keyword arguments
<code>def func(name)</code>	Function	Normal argument: matches any passed value by position or name
<code>def func(name=value)</code>	Function	Default argument value, if not passed in the call
<code>def func(*name)</code>	Function	Matches and collects remaining positional arguments in a tuple
<code>def func(**name)</code>	Function	Matches and collects remaining keyword arguments in a dictionary
<code>def func(*name, name)</code>	Function	Arguments that must be passed by keyword only in calls
<code>def func(*, name)</code>	Function	Arguments that must be passed by keyword only in calls
<code>def func(name, /)</code>	Function	Arguments that must be passed by position only in calls

The argument-matching modes listed in this table break down between function *calls* and *definitions* as follows:

In a function call (the first four rows of the table)

Simple values are matched to definition arguments by position, but using the *name=value* form tells Python to match arguments by name instead; these are called *keyword arguments*. Any number of **iterable* or ***dict* forms can be used in a call, allowing us to bundle many positional or keyword objects in iterables and mappings, respectively, and unpack them as separate, individual arguments when they are passed to the function.

In a function definition (the rest of the table)

A simple *name* is matched by position or name depending on how the caller passes it, but the *name=value* form specifies a *default value*. The **name* form collects any extra unmatched positional arguments in a tuple, and ***name* collects unmatched keyword arguments in a dictionary. In addition, any normal or defaulted argument names following a **name* or a bare *** are *keyword-only* arguments and must be passed by keyword in calls, and arguments preceding a */* are *positional-only* arguments that must *not* be passed by keyword name.

Of these, keyword arguments and defaults are probably the most commonly used in Python code. We've informally used both of these earlier in this book:

- We've already used *keywords* to specify options to the `print` function, but they are more general—keywords allow us to label any argument with its name, to make calls more explicit and informational.

- We met *defaults* earlier, too, as a way to pass in values from the enclosing function's scope, but they are also more general—they allow us to make any argument optional, providing its default value in a function definition.

As you'll see ahead, the combination of defaults in a function definition and keywords in a call further allows us to pick and choose which defaults to override per call.

In short, special argument-matching modes let you be fairly liberal about how many arguments must be passed to a function. If a function specifies defaults, they are used if you pass *too few* arguments. If a function uses the *** argument-collector forms, you can seemingly pass *too many* arguments; the *** names collect the extra arguments in data structures for processing within the function.

Argument Passing Details

If you choose to use and combine the special argument-matching modes, Python will ask you to follow some ordering rules among the modes' optional components. We're going to defer full and formal coverage of these until we've had a chance to observe these modes in action, but as some initial tips:

- In a function *call*, any number of positionals, keywords, and starred unpackings can be used, but positional arguments must precede keyword arguments and ***dict* unpackings, and **iterable* unpackings must precede ***dict* unpackings.
- In a function *definition*, arguments must appear in this order: any positional-only arguments; followed by any positional-or-keyword arguments; followed by the optional **name* positional collector or *** and any keyword-only arguments; followed by the optional ***name* keyword collector. Arguments can have optional defaults (*name=value*), but once a default is used, all arguments must use defaults up to a ***, after which keyword-only arguments allow defaults and nondefaults to be freely mixed.

If you mix arguments in any other order, you will get a syntax error because the combinations can be ambiguous. The steps that Python internally carries out to match arguments before assignment can roughly be described as follows:

1. Unpack all **args* at the call into nonkeyword arguments.
2. Unpack all ***args* at the call into keyword arguments.
3. Assign nonkeyword arguments by position.
4. Assign keyword arguments by matching names.
5. Collect extra nonkeyword arguments in the **name* tuple.
6. Collect extra keyword arguments in the ***name* dictionary.
7. Assign default values to unassigned arguments.

After arguments in call and definition are matched, Python checks to make sure each argument is passed just one value (or else an error is raised) and then assigns argument names to the objects passed to them and runs the function body.

The actual matching algorithm Python uses is a bit more complex, so we'll defer to Python's standard language manual for a more exact description. It's not required reading, but tracing Python's matching algorithm may help you to understand some convoluted cases, especially when modes are mixed.

We'll return to the ordering rules in function calls in definitions with higher fidelity after we've had a chance to meet all the players. Let's get started in the next section with the most common of the bunch.



But annotations are moot: Argument names in a `def` statement (only) can also have *annotation* values, specified as `name:annot`, or `name:annot=default` when defaults are present. This is simply additional syntax for arguments and does not augment or change argument-ordering rules. The function itself can also have an annotation value, given as `def f(...)->annot`, and Python attaches all annotation values to the function object. See the discussion of function annotation in [Chapter 19](#) for more details, and the section on their role in unused type hinting in [Chapter 6](#).

Keyword and Default Examples

Argument passing is simpler in code than the preceding descriptions may imply. First off, if you don't use any special matching syntax, Python matches names by *position* from left to right. For instance, if you define a function that requires three arguments, you must call it with three arguments:

```
>>> def f(a, b, c): print(a, b, c)

>>> f(1, 2, 3)
1 2 3
```

Here, we pass by position—`a` is matched to 1, `b` is matched to 2, and so on. This works like it does in most other programming languages.

Keywords

In Python, though, you can be more specific about what goes where when you call a function. Keyword arguments allow us to match by *name*, instead of by position. Using the same function definition but a different call:

```
>>> f(c=3, b=2, a=1)
1 2 3
```

The `c=3` in this call, for example, means send 3 to the argument named `c`. More formally, Python matches the name `c` in the call to the argument named `c` in the function definition, and then assigns the value 3 to that argument. The net effect of this call is the same as that of the prior call, but notice that the left-to-right order of the arguments no longer matters when keywords are used because arguments are matched by name, not by position.

It's even possible to combine positional and keyword arguments in a single call. In this case, all positionals are matched first from left to right in the definition, before keywords are matched by name:

```
>>> f(1, c=3, b=2)           # a gets 1 by position, b and c passed by name
1 2 3
```

When most people see this the first time, they wonder why one would use such a tool. Keywords typically have two roles in Python. First, they make your calls a bit more self-documenting (assuming that you use better argument names than `a`, `b`, and `c`!). For example, a call of this form:

```
func(title='Learning Python', edition=6, year=2024, python=3.12)
```

is much more meaningful than a call with three naked values separated by commas, especially in larger programs—the keywords serve as labels for the data in the call:

```
func(title='Learning Python', 6, 2024, 3.12)
```

The second major use of keywords occurs in conjunction with defaults, which we turn to next.

Defaults

We talked about defaults in brief earlier, when discussing nested function scopes. In short, defaults allow us to make selected function arguments *optional*; if not passed a value, the argument is assigned its default before the function runs. For example, here is a function that requires one argument and defaults two others:

```
>>> def f(a, b=2, c=3): print(a, b, c)           # a required, b and c optional
```

As noted earlier, defaults must appear after nondefaults at this point in a header (they can be mixed after a `*` as you'll see ahead). When we call this function, we must provide a value for `a`, either by position or by keyword; however, providing values for `b` and `c` is optional. If we don't pass values to `b` and `c`, they default to 2 and 3, respectively:

```
>>> f(1)                # Use defaults
1 2 3
>>> f(a=1)
1 2 3
```

If we pass two values, only `c` gets its default, and when passing three values, no defaults are used:

```
>>> f(1, 4)             # Override defaults
1 4 3
>>> f(1, 4, 5)
1 4 5
```

Finally, here is how the keyword and default features interact. Because they subvert the normal left-to-right positional mapping, keywords allow us to essentially skip over arguments with defaults:

```
>>> f(1, c=6)           # Choose defaults: b in the middle
1 2 6
```

Here, `a` gets 1 by position, `c` gets 6 by keyword, and `b`, in between, defaults to 2.

Be careful not to confuse the special *name=value* syntax in a function definition and a function call; in the *call* it means a match-by-name keyword argument, while in the *definition* it specifies a default for an optional argument. In both cases, this is not an assignment statement (despite its appearance); it is special syntax for these two contexts, which modifies the default argument-matching mechanics.

Combining keywords and defaults

Here is a slightly larger example that demonstrates keywords and defaults in action. In the following, the caller must always pass at least two arguments (to match `code` and `hack`), but the other two are optional. If they are omitted, Python assigns `script` and `app` to the defaults specified in the definition:

```
def func(code, hack, script=0, app=0):          # First 2 required
    print((code, hack, script, app))

func(1, 2)                                     # Output: (1, 2, 0, 0)
func(1, app=1, hack=0)                         # Output: (1, 0, 0, 1)
func(code=1, hack=0)                           # Output: (1, 0, 0, 0)
func(script=1, hack=2, code=3)                 # Output: (3, 2, 1, 0)
func(1, 2, 3, 4)                              # Output: (1, 2, 3, 4)
```

Notice again that when keyword arguments are used in the call, the order in which the arguments are listed doesn't matter; Python matches by name, not by position. The caller must supply values for `code` and `hack`, but they can be given by position or by name.

Also keep in mind that all the special definition-side argument-matching syntax we're exploring in this chapter works the same in `def` and `lambda`, though most examples use the former, and the latter returns results implicitly:

```
>>> func = lambda code, hack, script=0, app=0: (code, hack, script, app)
>>> func(1, 2)
(1, 2, 0, 0)
>>> func(script=1, hack=2, code=3)
(3, 2, 1, 0)
```



Beware mutable defaults: As noted in the prior chapter, if you code a default to be a mutable object (e.g., `def f(a=[])`), the same, *single* mutable object is reused every time the function is later called—even if it is changed in place within the function. The net effect is that the argument's default retains its value from the prior call and is not reset to its original value coded in the function header. To reset anew on each call, move the assignment into the function body instead. Mutable defaults allow state retention, but this is often an unpleasant surprise. Since this is such a common trap, we'll postpone further exploration until this part's “gotchas” list at the end of [Chapter 21](#).

Arbitrary Arguments Examples

The last two argument-matching extensions, `*` and `**`, are designed to support *any number* of arguments. Both can appear in either the function definition or a function call, and they have related purposes in the two locations.

Definitions: Collecting arguments

The first use, in the function definition, collects unmatched *positional* arguments into a tuple:

```
>>> def f(*args): print(args)
```

When this function is called, Python *collects* all the positional arguments into a new *tuple* and assigns the variable `args` to that tuple. Because it is a normal tuple object, it can be indexed, stepped through with a `for` loop, and so on:

```
>>> f()
()
>>> f(1)
(1,)
>>> f(1, 2, 3, 4)
(1, 2, 3, 4)
```

The `**` feature is similar, but it only works for *keyword* arguments—it collects them into a new *dictionary*, which can then be processed with normal dictionary tools. In a sense, the `**` form allows you to convert from keywords to dictionaries, which you can then step through with `keys` calls, dictionary iterators, and the like (this is roughly what the `dict` call does when passed keywords, but it *returns* the new dictionary):

```
>>> def f(**args): print(args)

>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

Finally, function definitions can combine normal arguments, the `*`, and the `**` to implement wildly flexible call signatures. For instance, in the following, 1 is passed to `a` by position, 2 and 3 are collected into the `pargs` positional tuple, and `x` and `y` wind up in the `kargs` keyword dictionary:

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)

>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'x': 1, 'y': 2}
```

Notice that the dictionary's keys are ordered here: as of Python 3.6, the `**` collector's keys preserve the order in which keyword arguments were passed to the function. This relies on the insertion order of keys in dictionaries at large (see [Chapter 8](#) for a refresher if you've forgotten what that means).

Functions with both `*` and `**` may be rare, but they show up in functions that need to support multiple call patterns (for backward compatibility, for instance). In fact, these features can be combined in even more complex ways that may seem ambiguous at first glance—an idea we will revisit later in this chapter when we lock down ordering rules. First, though, let's see what happens when `*` and `**` are coded in function *calls* instead of definitions.

Calls: Unpacking arguments

It turns out that we can use the `*` syntax when we call a function, too. In this context, its meaning is the inverse of its meaning in the function definition—it *unpacks* a collection of arguments, rather than building a collection of arguments. For example, we can pass four arguments to a function in a tuple or other iterable, and let Python unpack them into individual positional arguments:

```
>>> def func(a, b, c, d): print(a, b, c, d)

>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)                                # Same as func(1, 2, 3, 4)
1 2 3 4
```

Similarly, the `**` syntax in a function call unpacks a dictionary or other mapping of key/value pairs into separate keyword arguments:

```
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)                                # Same as func(a=1, b=2, c=3, d=4)
1 2 3 4
```

Moreover, we can use other iterables like lists and other dictionary syntax like `dict`, and we may combine star, positional, and keyword arguments in the call in very flexible ways:

```
>>> func(*(1, 2), **{'d': 4, 'c': 3})            # Same as func(1, 2, d=4, c=3)
1 2 3 4
>>> func(1, *[2, 3], **dict(d=4))              # Same as func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, c=3, *[2], **{'d': 4})            # Same as func(1, 2, c=3, d=4)
1 2 3 4
>>> func(1, *(2, 3), d=4)                      # Same as func(1, 2, 3, d=4)
1 2 3 4
```

```
>>> func(1, *[2], c=3, **dict(d=4))      # Same as func(1, 2, c=3, d=4)
1 2 3 4
```

While the preceding serves to demo the possibilities, its use of stars and literals is overkill when arguments are known; more typical code would build up argument collections ahead of the call and unpack by names:

```
>>> pargs, kargs = (1, 2), dict(d=4, c=3)
>>> func(*pargs, **kargs)
1 2 3 4
```

Star unpacking is convenient when you cannot predict the number of arguments that will be passed to a function when you write your script; you can build up a collection of arguments at runtime instead and call the function generically this way. Here again, though, don't confuse the `*/**` starred-argument syntax in the function definition and the function call—in the *definition* it collects any number of arguments, while in the *call* it unpacks any number of arguments. In both, one star means positionals, and two applies to keywords.

Finally, as of Python 3.5, and as noted in [Chapter 11](#)'s sidebar “The Many Stars of Python” on page 270, we can even use *multiple* `*` and `**` items in calls to unpack multiple iterables and mappings, respectively. The `*` unpacks into positional arguments, and `**` into keyword arguments, though single stars must precede double stars, per formal rules coming up ahead. Again, the following uses literals to demo, but these would usually be names assigned to prebuilt values:

```
>>> def func(a, b, c, d): print(a, b, c, d)

>>> func(*[1], *(2,), **dict(c=3), **{'d': 4})      # *Positionals + **keywords
1 2 3 4
>>> func(*[1], *(2,), *[3, 4])                    # All *positionals
1 2 3 4
>>> func(**dict(a=1, b=2), **dict(c=3), **{'d': 4}) # All **keywords
1 2 3 4
>>> func(*[1], 2, **dict(c=3), d=4)                # All call modes at once!
1 2 3 4

>>> func(*[1], **dict(b=2, c=3), *[4])
SyntaxError: iterable argument unpacking follows keyword argument unpacking
```



Unpacking generality—and inconsistency: As previewed in [Chapter 14](#), the `*` form in a call is an *iteration tool*, so it accepts any iterable object, not just tuples or other sequences used in examples here. For instance, iterables like `zip` and `range` work after a `*` too and unpack into individual arguments, and file objects automatically read and unpack their lines:

```
func(*range(4))      # Same as func(0, 1, 2, 3)
func(*open('filename')) # Read+pass lines as arguments
```

Watch for more examples of this utility in [Chapter 20](#), after we study generators.

On the downside, stars also come with *inconsistencies*. For one, this generality holds true only for *calls*—a `*` unpacking in a call allows any iterable, but a `*` in a function *definition* always collects extra arguments into a *tuple*. Moreover, this collection behavior in definitions is similar in spirit and syntax to the `*` in the extended-unpacking *assignment* forms we met in [Chapter 11](#) (e.g., `x, *y = z`), but that star usage always creates *lists*, not tuples. Again: these are different rules for different tools—despite the same syntax.

Why arbitrary arguments?

The prior section's examples may seem academic (if not downright esoteric), but they are used more often than you might expect. Some programs need to call arbitrary functions in a generic fashion, without hardcoding their names or arguments ahead of time. In fact, the real power of the special starred-unpacking call syntax is that you don't need to know how many arguments a function call requires before you write a script. For example, you can use `if` logic to select from a set of functions and argument lists, and call any of them generically (functions here are hypothetical):

```
if sometest:
    action, args = func1, (1,)          # Call func1 with one arg in this case
else:
    action, args = func2, (1, 2, 3)     # Call func2 with three args instead
...
action(*args)                          # Dispatch generically
```

This leverages both the `*` form and the fact that functions are objects that may be both referenced by, and called through, any variable. More generally, this unpacking call syntax is useful anytime you cannot predict the arguments list. If your user selects an arbitrary function via a user interface, for instance, you may be unable to hardcode a function call when writing your script. To work around this, simply build up the arguments list with sequence operations, and call it with starred-argument syntax to unpack the arguments:

```
args = (2,3)
args += (4,)
...
func3(*args)
```

Because the arguments list is passed in as a tuple here, the program can build it at runtime. This technique also comes in handy for functions that test or time other functions. For instance, the code in [Example 18-1](#) supports any function with any arguments by passing along whatever arguments were sent in (see *tracer0.py* in the example package).

Example 18-1. *tracer0.py*

```
def tracer(func, *pargs, **kargs):      # Accept arbitrary arguments
    print('calling:', func.__name__)
    return func(*pargs, **kargs)       # Pass along arbitrary arguments

def func(a, b, c, d):
    return a + b + c + d

print(tracer(func, 1, 2, c=3, d=4))
```

This code uses the built-in `__name__` attribute attached to every function (as you might expect, it's the function's name string), and uses stars to collect and then unpack the arguments intended for the traced function. In other words, when this code is run, arguments are intercepted by the tracer and then *propagated* with unpacking call syntax:

```
$ python3 tracer0.py
calling: func
10
```

For another example of this technique, see the preview near the end of the preceding chapter, where it was used to reset the built-in `open` function (though it probably makes more sense to you now). We'll code additional examples of such roles later in this book; see especially the sequence timing examples in

Chapter 21 and the various *decorator* utilities we will code in Chapter 39 (after a preview in the next chapter). It's a common technique in general tools.

Keyword-Only Arguments

Besides accepting keyword (i.e., pass-by-name) arguments in general, function *definitions* can also specify that some arguments must always be passed by keyword only and will never be filled in by a positional argument. This extension, known as *keyword-only arguments*, can be useful if we want a function to both process any number of arguments and accept possibly optional configuration options.

Syntactically, keyword-only arguments are coded as named arguments that may appear after **name* in the arguments list. All such arguments must be passed using keyword syntax in the call. For example, in the following, *a* may be passed by name or position, *b* collects any extra positional arguments, and *c* must be passed by keyword only:

```
>>> def kwnonly(a, *b, c): print(a, b, c)

>>> kwnonly(1, 2, c=3)
1 (2,) 3
>>> kwnonly(a=1, c=3)
1 () 3
>>> kwnonly(1, 2, 3)
TypeError: kwnonly() missing 1 required keyword-only argument: 'c'
```

If we don't need to collect arbitrary positionals, we can also use a bare *** character by itself in the arguments list to introduce keyword-only arguments. This indicates that a function does not accept a variable-length argument list but still expects all arguments following the *** to be passed as keywords. In the next function, *a* may be passed by position or name again, but *b* and *c* must be keywords, and no extra positionals are allowed:

```
>>> def kwnonly(a, *, b, c): print(a, b, c)

>>> kwnonly(1, c=3, b=2)
1 2 3
>>> kwnonly(c=3, b=2, a=1)
1 2 3
>>> kwnonly(1, 2, 3)
TypeError: kwnonly() takes 1 positional argument but 3 were given
>>> kwnonly(1)
TypeError: kwnonly() missing 2 required keyword-only arguments: 'b' and 'c'
```

You can still use defaults for keyword-only arguments, even though they appear after the *** in the function. In the following, *a* may be passed by name or position, and *b* and *c* are optional but must be passed by keyword if used:

```
>>> def kwnonly(a, *, b='code', c='app'): print(a, b, c)

>>> kwnonly(1)
1 code app
>>> kwnonly(1, c='hack')
1 code hack
>>> kwnonly(a='py')
py code app
>>> kwnonly(c=3, b=2, a=1)
1 2 3
>>> kwnonly(1, 2)
TypeError: kwnonly() takes 1 positional argument but 2 were given
```

In fact, keyword-only arguments with defaults are optional, but those without defaults effectively become *required keywords* for the function—like `b` in the following:

```
>>> def kwonly(a, *, b, c='hack'): print(a, b, c)

>>> kwonly(1, b='code')
1 code hack
>>> kwonly(1, c='code')
TypeError: kwonly() missing 1 required keyword-only argument: 'b'
>>> kwonly(1, 2)
TypeError: kwonly() takes 1 positional argument but 2 were given
```

As noted earlier, keyword-only arguments also allow defaults and nondefaults to be *mixed*, unlike their otherwise more flexible cohorts coded before the optional `*`—an ostensible inconsistency we’ll return to later:

```
>>> def kwonly(a, *, b=2, c, d=4): print(a, b, c, d)

>>> kwonly(1, c=3)
1 2 3 4
>>> kwonly(1, c=5, b=6)
1 6 5 4
>>> kwonly(1)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes 1 positional argument but 3 were given

>>> def badnix(b=2, c, d=5): ...
SyntaxError: parameter without a default follows parameter with a default
```

Finally, note that keyword-only arguments must be specified after a single star, not two—nothing can appear after the `**args` arbitrary-keywords form, and, unlike `*`, a `**` can’t appear by itself in the arguments list. More generally, keyword-only arguments must be coded *between* the `*` and the optional `**`, and an argument that appears before `*` is a possibly default argument that can be passed by position or keyword, not keyword-only:

```
>>> def kwonly(a, **kargs, b, c):
SyntaxError: arguments cannot follow var-keyword argument
>>> def kwonly(a, **, b, c):
SyntaxError: invalid syntax
>>> def mixed(a, *b, **d, c=6):
SyntaxError: arguments cannot follow var-keyword argument
```

These failures will make more sense after we get formal about argument ordering rules later in this chapter. They may also appear to be worst cases in the artificial examples here, but they can come up in real practice, especially for people who write libraries for other Python programmers to use—which leads to the next point.

Why keyword-only arguments?

So why care about keyword-only arguments? In short, they make it easier to allow a function to accept both any number of positional arguments to be processed, and configuration options passed as keywords. While their use is optional, without keyword-only arguments extra work may be required to provide defaults for such options and to verify that no superfluous keywords were passed.

Imagine a function that processes a set of passed-in objects and allows a tracing flag to be passed:

```
process(X, Y, Z)                # Use flag's default
process(X, Y, notify=True)      # Override flag default
```


Without keyword-only arguments we have to use both `*args` and `**args` and manually inspect the keywords, but with keyword-only arguments less code is required. The following guarantees that no positional argument will be incorrectly matched against `notify` and requires that it be a keyword if passed:

```
def process(*args, notify=False): ...
```

Since we're going to explore a more realistic example of this later in this chapter, in [“Example: Rolling Your Own Print” on page 448](#), we'll postpone the rest of this story until then. For an additional example of keyword-only arguments in action, see the upcoming iteration-options timing case study in [“Timer Module: Take 2” on page 531](#).

Positional-Only Arguments

Beginning with Python 3.8, function *definitions* may also include a `/` in the arguments list to designate that all arguments preceding it (i.e., to its left) must be passed by *position*, not by keyword-argument name. Though arguably ad hoc on first sighting in an arguments list, this notation was being used in documentation for built-in functions that did not accept keywords; making it available to function coders as part of Python's syntax was deemed a logical extension for library developers who may not want to expose argument names for use by clients as keywords.

To demo, the following function specifies that `a` and `b` must be passed by position, though `c` is more flexible:

```
>>> def mostlypos(a, b, /, c): print(a, b, c)

>>> mostlypos(1, 2, 3)
1 2 3
>>> mostlypos(1, 2, c=3)
1 2 3
```

Passing either of the first two arguments by keyword, however, fails:

```
>>> mostlypos(1, b=2, c=3)
TypeError: mostlypos() got some positional-only arguments passed as keyword arguments: 'b'

>>> mostlypos(c=3, b=2, a=1)
TypeError: mostlypos() got some positional-only arguments passed as keyword arguments: 'a, b'
```

To define a function that allows *only* positional arguments, simply code the slash at the end:

```
>>> def allpos(a, b, c, /): print(a, b, c)

>>> allpos(1, 2, 3)
1 2 3
>>> allpos(1, 2, c=3)
TypeError: mostlypos() got some positional-only arguments passed as keyword ...
```

As you should expect, the slash works the same in a `lambda` argument list:

```
>>> f = lambda a, b, /, c: print(a, b, c)

>>> f(1, 2, c=3)
1 2 3
>>> f(1, b=2, c=3)
TypeError: <lambda>() got some positional-only arguments passed as keyword ...
```

And functions can *combine* positional- and keyword-only arguments to be as rigid as they wish:

```
>>> def combo(a, b, /, *, c, d): print(a, b, c, d)

>>> combo(1, 2, c=3, d=4)
1 2 3 4
>>> combo(1, 2, 3, 4)
TypeError: combo() takes 2 positional arguments but 4 were given
>>> combo(a=1, b=2, c=3, d=4)
TypeError: combo() got some positional-only arguments passed as keyword ...
```

It's up to you to ponder whether or not use cases for this syntax justify its convolution of function definitions that follows in the next section. Given that Python users somehow got by without it for over three decades, though, this seems a tough sell. For more on the rationale and usage of the positional-only slash, see Python's standard manuals. Also watch for an example in [Chapter 21](#) that uses it to avoid name clashes—and may or may not be compelling.

Argument Ordering: The Gritty Details

So far, we've been fairly loose about the rules surrounding argument-matching tools, because they don't crop up very often in the simpler usage patterns of typical code. In more sophisticated roles, though, you need to verify that your code follows Python's expectations. Now that we've seen all of its subjects, the ordering rules for function arguments can finally be summarized in full. While function *definitions* and *calls* share some similar syntax, their rules are completely different, owing to their different roles. Let's take a more formal look at both.

Definition Ordering

In function *definitions*, argument lists are enclosed in parentheses in a `def` statement and coded before a colon in a `lambda` expression, but follow the same format in both. In short, they consist of four optional parts that must appear in the following order, where *position* means a simple value in calls, and *keyword* means a `name=value` pair:

1. One or more arguments that must be passed by *position* only, followed by a single `/`
2. Any number of arguments that can be passed by either *position* or *keyword*
3. A single `*` by itself or a single `*name` positional-argument *collector*, optionally followed by any number of arguments that must be passed by *keyword* only
4. A single `**name` keyword-argument *collector*

In all cases, individual arguments, including a bare `/` or `*`, are separated by commas. In addition, any nonstarred argument name can have a `name=expression` default, but all names must have defaults after the first that does, up to the `*` (keyword-only argument runs following a star may freely mix default and nondefault names).

Inherent in this ordering, positional-only arguments must appear first, `*name` ends positional-argument runs and collects unmatched positionals, and `**name` ends the entire arguments list and collects unmatched keywords. As noted earlier, the `**name` collector's keys retain the order in which keyword arguments were passed to the function.

Formal definition

More concisely, the ordering of arguments in function definitions can be defined as follows, where *-or-* is notation for a choice and `[]` encloses an optional part (neither is part of the actual code you type):

```
def name(arguments-list): statements
    lambda arguments-list: expression

arguments-list =
    [positional-only-arguments, /]
    [positional-or-keyword-arguments]
    [* -or- *positional-collector, [keyword-only-arguments]]
    [**keyword-collector]
```

As a real example, the following defines a function with all these parts in actions:

```
>>> def f(a, /, b, c=3, *ps, e=4, **ks):
    print(f'{a=}, {b=}, {c=}, {ps=}, {e=}, {ks=}')

>>> f(0, 1, 2, 3, 4, e=5, f=6, g=7)
a=0, b=1, c=2, ps=(3, 4), e=5, ks={'f': 6, 'g': 7}
```

Boundary cases

As consequences of the ordering in function definitions, `/` must precede a star, and the double-star keyword collector must be coded last:

```
>>> def f(a, *ps, /, b): pass
SyntaxError: / must be ahead of *

>>> def f(a, **ks, b): pass
SyntaxError: arguments cannot follow var-keyword argument
```

Though enforced separately from basic argument-ordering rules, once a *default* is used in a definition, all subsequent arguments must also have defaults—including those following the positional-only / delimiter:

```
>>> def f(a, b, c=3, d, e): pass
SyntaxError: parameter without a default follows parameter with a default

>>> def f(a, b=2, /, c): pass
SyntaxError: parameter without a default follows parameter with a default
```

However, this constraint applies only to arguments preceding a `*` or **name*—defaults and nondefaults *can* be mixed freely in keyword-only arguments, which seems inconsistent, though a case for sanity could be made here on the grounds that keyword-only arguments never match by position, which reduces ambiguity of defaults:

```
>>> def f(a, *, x=1, y): ...      # OK: x and y must be keywords

>>> def f(a=1, b, *, x=1): ...    # Not OK: a and b may be positional
SyntaxError: parameter without a default follows parameter with a default
```

Not shown here, `def` can also be preceded by a *decorator* (e.g., `@value`), and, as noted earlier, any of its arguments may be followed by *annotations* (e.g., `:value`)—extensions we'll explore in the next chapter and later in this book. Neither of these impact argument ordering or matching, and neither work in `lambda` due to its limited syntax.

Calls Ordering

On the other side of the fence, the syntax is similar but the rules differ. In function *calls*, all positional arguments must precede all keyword arguments, and any number of starred *unpackings* can be mixed in with individual values: one star unpacks into multiple positional arguments, two unpacks into keywords, and all positional arguments and one-star unpackings must precede all two-star unpackings.

In more detail, function (and by extension, method) calls consist of three optional parts in the following order:

1. Any combination of one or more *expression* positional arguments, and **expression* iterable unpackings transformed into positional arguments
2. Any combination of one or more *name=expression* keyword arguments, and **expression* iterable unpackings transformed into positional arguments
3. Any combination of one or more *name=expression* keyword arguments, and ***expression* mapping unpackings transformed into keyword arguments

In all cases, all arguments, starred or otherwise, are separated by commas.

Formal definition

More concisely again, the ordering of arguments in function calls can be defined as follows, where *function* is an expression that evaluates to a callable object, and *-or-* and *[]* here again mean a choice and optional part, respectively (they're not part of the code you type):

```
function([positional-values -or- *iterable-positional-unpackings]
         [keyword-arguments -or- *iterable-positional-unpackings]
         [keyword-arguments -or- **mapping-keyword-unpackings])
```

As a concrete example, the following passes a variety of positional, keyword, and unpacking arguments (this code serves as a demo here, but is not exactly the sort of thing you should strive to craft in practice!):

```
>>> def f(a, b, c, d, e, f, g, h, i):
    print(a, b, c, d, e, f, g, h, i)

>>> f(*[1], 2, *[3], 4, f=6, *[5], **dict(g=7), h=8, **{'i': 9})
1 2 3 4 5 6 7 8 9
```

Boundary cases

As one consequence of the argument ordering in calls, once you use a *keyword* argument, you can no longer use any unstarred positionals—all subsequent arguments must also be keywords, or single or double stars:

```
>>> f(1, 2, c=3, 4)
SyntaxError: positional argument follows keyword argument unpacking
```

This is similar to defaults in definitions—but not exactly. Again, try not to conflate function definitions and calls. Despite their reuse of similar syntax, it has very different roles and rules in these tools. The *** and *=*, for example, are used for unpacking and keywords in calls, but mean collection and defaults in definitions.

Once you code a *double star*, both positionals and single stars are also out of the game, though this seems inconsistent too—single stars can be freely mixed with keyword arguments (which is what a

double star unpacks into), and keyword arguments cannot be mixed with positionals (which is what a single star unpacks into):

```
>>> f(1, 2, **{'d': 3}, 4)
SyntaxError: positional argument follows keyword argument unpacking
>>> f(1, 2, **{'d': 3}, *[4])
SyntaxError: iterable argument unpacking follows keyword argument unpacking

>>> f(1, 2, d=3, *[4])    # OK, but why? - like first error above
>>> f(1, 2, d=3, 4)       # Not OK, but why? - like preceding line
SyntaxError: positional argument follows keyword argument
```

Perspective

And if this is starting to make your head spin, it probably should. Python's argument-matching rules have been accumulated over time to incorporate new convolutions, and they are complex and perhaps even kludgy. The first rule of programming applies to function definitions and calls as everywhere else: keep it simple, unless it has to be complex. If you have to agonize over argument-ordering rules to understand code, it's probably time to reevaluate priorities.

Example: The min Wakeup Call

OK—it's time for something more realistic. To make the concepts here more concrete, the rest of this chapter works through a set of examples that demonstrate practical applications of argument-matching tools. First up is an exercise borrowed from live classes and used to rouse learners like you starting to succumb to the knottiness of argument rules.

Here's the problem statement: suppose you're asked to code a function that is able to compute the *minimum value* from an arbitrary set of arguments, which may be arbitrary sorts of objects. That is, the function should accept *zero or more* arguments—as many as you wish to pass. Moreover, the function should work for all kinds of Python object *types*: numbers, strings, lists, lists of lists, files, and even `None`. To keep this fair, you don't need to support *dictionaries* or *mixed* nonnumeric types, because neither supports direct comparisons, per Chapters 8 and 9.

The first requirement provides a natural example of how the `*` feature can be put to good use—we can handle arbitrary arguments by collecting them in a tuple, and stepping over each with a simple `for` loop. The second part of the problem definition is easy in Python: because nearly every object type supports comparisons, we don't have to specialize the function per type (an application of *polymorphism*); we can simply compare objects blindly and let Python worry about what sort of comparison to perform according to the objects being compared.

Full Credit

The following script file, *mins.py* in [Example 18-2](#), shows four ways to code this operation (some of which were suggested by students in a group exercise designed to prevent post-lunch napping):

- The first function fetches the first argument from its `args` tuple, and traverses the rest by slicing off the first (there's no point in comparing the first object to itself, especially if it might be a large structure).
- The second version lets Python pick off the first and rest of the arguments automatically, and so avoids an index and slice; the code is simpler, and may be faster (though it would take many calls to matter).

- The third converts from a tuple to a list with the built-in `list` call and employs the list sort method: the first item has lowest value after an ascending-value sort.
- The fourth sorts too, but skips the list conversion (and two lines) by using the sorted built-in function.

Python sorting tools are coded in C, so they can be quicker than the other approaches at times, but the linear scans of the first two techniques may often make them faster.¹

Example 18-2. *mins.py*

"Find minimum value among all passed arguments of comparable types"

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res

def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first

def min3(*args):
    tmp = list(args)
    tmp.sort()
    return tmp[0]

def min4(*args):
    return sorted(args)[0]

for func in (min1, min2, min3, min4):
    print(func.__name__ + '...')
    print(func(3, 4.0, 1, 2))
    print(func('bb', 'aa'))
    print(func([2, 2], [1, 1], [3, 3]))
    print(func(*'hack'))
```

Test all 4 functions
Mixed numerics
Strings: code points
Lists: recursive
Unpacked characters

This script's testing code uses the `__name__` attribute we met earlier, along with a `for` loop to run each function one at a time (remember, functions are objects that work in a tuple too). All four solutions produce the same result when the file is run, so we'll list just the first's output here. Run this file live, or

¹ Actually, it's complicated. CPython's sort (used by both `list.sort` and `sorted`) is coded in C and uses a heavily optimized algorithm that attempts to take advantage of partial ordering in the items to be sorted. Still, sorting is an inherently busy operation (it must chop up the sequence and put it back together many times), and the other versions simply perform one linear left-to-right scan. This suggests that sorting may be quicker if the arguments are partially *ordered*, but is likely to be slower otherwise. Even so, Python performance changes regularly; the fact that sorting is implemented in the C language can help greatly; and the speed difference may not matter in many programs. For an exact analysis, you should time the alternatives with the `time` or `timeit` modules—you'll see how soon in [Chapter 21](#), but file *mins-timings.txt* in the examples package demos the idea if you can't wait. The gist: in CPython, the nonsort mins are faster for random arguments, but slower for ordered—today!

import it as a module and type a few calls to its function interactively to experiment with them on your own (see [Chapter 3](#) for tips on both modes):

```
$ python3 mins.py
min1...
1
aa
[1, 1]
a
...and the same for others...
```

Notice that none of these four variants tests for the case where *no* arguments are passed in. They could, but there's probably no point in doing so here—in all four solutions, Python will automatically raise an exception to signal the error if no arguments are sent. The first variant raises an exception when we try to fetch argument 0, the second when Python detects an argument list mismatch, and the third and fourth when we try to return item 0 post sort.

This is exactly what we want to happen—because these functions support any object (including None), there is no value that we could pass back to designate an error, so we may as well let the exception be raised. There are exceptions to this exceptions rule (e.g., you might test for errors yourself if you'd rather avoid actions that run before reaching the code that triggers an error automatically). But in general—and especially when errors aren't common—it's better to assume that arguments will work in your functions' code, and let Python raise errors for you when they do not.

Bonus Points

You can get bonus points here for changing these functions to compute the arguments' *maximum*, rather than minimum, value. This one's trivial: the first two versions only require changing < to >, and the last two simply require that we return item [-1] instead of [0]. For an extra point, be sure to mod the function name to "max" as well (though this part is strictly optional).

True curve busters might also note that it's possible to generalize a *single* function to compute either a minimum *or* a maximum value, by evaluating comparison expression strings with a tool like the eval built-in function (described in Python's library manual, and at various appearances here, especially its note in [Chapter 10](#)), or by passing in an arbitrary comparison function. [Example 18-3](#) shows how to implement the latter scheme for one of the coding options.

Example 18-3. minmax.py

```
"Find minimum -or- maximum value of arguments"

def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y           # See also: lambda, eval
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))  # Self-test code
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

Running this script prints both minimum and maximum, per its self-test code at the end:

```
$ python3 minmax.py
1
6
```

Again, functions are just another kind of object, which allows them to be passed into other functions as done here. To make this a `max` (or other) function, for example, we simply pass in the right sort of test function to `minmax`. This may seem like extra work, but the main point of *generalizing* functions this way—instead of cutting and pasting to change just a single character—is that we’ll only have one version to change in the future, not two.

The Punch Line

Of course, all this was just a coding exercise. There’s really no reason to write `min` or `max` functions, because both are built-ins in Python! We met them briefly in [Chapter 5](#) in conjunction with numeric tools, and again in [Chapter 14](#) when exploring iteration contexts. The built-in versions work almost exactly like ours, but they’re coded in C for optimal speed and accept either a single iterable or multiple arguments. Still, though it’s superfluous in this context, the general coding pattern we used here might be useful in other scenarios.

Example: Generalized Set Functions

Our next example also demos special argument-matching modes at work. At the end of [Chapter 16](#), we wrote a function that returned the intersection of two sequences (really, it picked out items that appeared in both). [Example 18-4](#) codes an augmented version that intersects an *arbitrary* number of sequences (one or more) by using the argument-matching form `*args` to collect all the passed-in arguments. Because the arguments come in as a tuple, we can process them in a simple `for` loop. Just for fun, we’ll code a union function that also accepts an arbitrary number of arguments to collect items that appear in *any* of the operands.

Example 18-4. *inter2.py*

```
"""
Implement intersection and union for one or more arguments.
Inputs may be any sort of iterable that supports multiple in
tests, and results are always lists. This intersect avoids
duplicates in results by in test, but may be slow: improve me.
"""

def intersect(*args):
    res = []
    for x in args[0]:
        if x in res: continue
        for other in args[1:]:
            if x not in other: break
        else:
            res.append(x)
    return res

def union(*args):
    res = []
    for seq in args:
        for x in seq:
```



```

        if not x in res:
            res.append(x)          # Add new items to result
    return res

```

Because these tools are potentially worth reusing (and are too big to retype interactively), we'll store the functions in a *module* file called *inter2.py*. Again, if you're unsure about how modules and imports work, see the introduction in [Chapter 3](#), or stay tuned for in-depth coverage in [Part V](#). This chapter's module usage is simple, but per [Chapter 3](#), be sure to launch your REPL in the same *folder* as the file, so imports can find it.

Like [Chapter 16](#)'s original *intersect*, both of the functions in this module work on any kind of sequence. Here they are live at the REPL, processing strings, mixed types, and more than two sequences:

```

$ python3
>>> from inter2 import intersect, union
>>> s1, s2, s3 = 'HACKK', 'CODE', 'CASH'

>>> intersect(s1, s2), union(s1, s3)      # Two operands
(['C'], ['H', 'A', 'C', 'K', 'S'])

>>> intersect([1, 2, 3, 4], (1, 4))        # Mixed types
[1, 4]

>>> intersect(s1, s2, s3)                 # Three operands
['C']

>>> union(s1, s2, s3)
['H', 'A', 'C', 'K', 'O', 'D', 'E', 'S']

```

Testing the Code

To test more thoroughly, the following continues our REPL session to code a function that applies the two tools to arguments in different orders using a simple *shuffling* technique that we saw in [Chapter 13](#)—it slices to move the first to the end on each loop, uses a *** to unpack arguments, and sorts so results are comparable. Notice that arguments for the function are sent as a sequence (not discrete items), and the trace configuration option is keyword-only here:

```

>>> def tester(func, items, *, trace=True):
    for i in range(len(items)):
        items = items[1:] + items[:1]    # Move front item to back
        if trace: print(items)
        print(sorted(func(*items)))       # Test with reordered items

>>> tester(intersect, (s1, s2, s3))      # Use strings from prior listing
('CODE', 'CASH', 'HACKK')
['C']
('CASH', 'HACKK', 'CODE')
['C']
('HACKK', 'CODE', 'CASH')
['C']

>>> tester(union, (s1, s2, s3), trace=False)
['A', 'C', 'D', 'E', 'H', 'K', 'O', 'S']
['A', 'C', 'D', 'E', 'H', 'K', 'O', 'S']
['A', 'C', 'D', 'E', 'H', 'K', 'O', 'S']

>>> tester(intersect, (s1, s2, s3), trace=False)

```

```
['C']  
['C']  
['C']
```

Two context notes here: first, because *duplicates* won't appear in these intersection and union functions, they qualify as set operations mathematically, but may not be optimal in term of speed. Still, there's not much point in improving this demo's code—intersection and union, like `min` and `max`, are built-in operations today: the *set* object we explored in [Chapter 5](#) does intersection and union with `&` and `|`, and has methods that take multiple operands too. Hence, optimizing this code is left as suggested exercise, but see *inter3.py* in the examples package for pointers.

Second, the argument scrambling in the tester here doesn't generate all possible argument orders (that would require a full permutation, and 6 orderings for 3 arguments), but suffices to check if argument order impacts results. As a preview, though, the tester would be simpler and more flexible if it delegated scrambling to a reusable *function*. Watch for this revision in [Chapter 20](#), after we've explored how to code user-defined *generators*. We'll also recode set tools one last time in [Chapter 32](#) and a solution to a [Part VI](#) exercise, as classes that add them to the list object as methods.

Example: Rolling Your Own Print

To close out the chapter, let's look at one last example of argument matching at work: this section uses both `*` and `**` arbitrary-arguments collectors up front, and keyword-only arguments later, to emulate most of what Python's `print` function does. Like the preceding examples, there's no urgent reason to code tools that Python provides. Also like the others, though, this is both instructive and may be a basis for custom variants of built-in tools.

For both purposes, the module file in [Example 18-5](#) does roughly the same job as `print` in a small amount of reusable and modifiable code, by building and routing the print string per configuration arguments.

Example 18-5. *print3.py*

```
"""  
Emulate most of the Python 3.X print function as customizable code.  
Call signature: print3(*args, sep=' ', end='\n', file=sys.stdout).  
"""  
  
import sys  
  
def print3(*args, **kwargs):  
    sep = kwargs.get('sep', ' ')          # Keyword arg defaults  
    end = kwargs.get('end', '\n')  
    file = kwargs.get('file', sys.stdout)  
    output = ''                          # Build+print a string  
    first = True  
    for arg in args:  
        output += ('' if first else sep) + str(arg)  
        first = False  
    file.write(output + end)
```

Notice that this module's docstring uses a *raw string* to retain its backslash for help (per [Chapter 15](#)). Also note that this module's function need not be called “print3” because “print” is a built-in but not a reserved word, but using a different name avoids inadvertently hiding the built-in. To test it, import this into another file or the interactive prompt, and use it like the `print` built-in. [Example 18-6](#) codes a test script that imports our printer as a demo.

Example 18-6. *test-print3.py*

```
from print3 import print3
print3(1, 2, 3)                # Defaults
print3(1, 2, 3, sep='')        # Suppress separator
print3(1, 2, 3, sep='...')     # Custom separator
print3(1, [2], (3,), sep='...') # Various object types

print3(4, 5, 6, sep='', end='') # Suppress newline
print3(7, 8, 9)                # Finish line
print3()                       # Blank line

import sys
print3(1, 2, 3, sep='?', end='.\n', file=sys.stderr) # Redirect to stream

print3('LP6E was here', file=open('log.txt', 'w'))    # Redirect to a file
print3(open('log.txt').read())
```

When this is run, our `print3` produces the same results as the `print` built-in. Fine points here: *stderr* goes to your console by default, and it's OK to use a *dash* in this script's name because it's run, not imported (again, we'll be focusing on such module details in this book's next part):

```
$ python3 test-print3.py
1 2 3
123
1...2...3
1...[2]...(3,)
4567 8 9

1?2?3.
LP6E was here
```

As usual, the generality of its toolset allows us to prototype or develop concepts in the Python language itself. In this case, argument-matching tools are as flexible in Python code as they are in Python's internal implementation.

Using Keyword-Only Arguments

Our print emulator works, but has a minor flaw baked in: it assumes that all positional arguments are to be printed, and all keywords are for options only. Any extra keyword arguments are silently ignored, and neither printed nor reported. A call like the following, for instance, will ignore the extra—and likely erroneous—`sap` argument:

```
$ python3
>>> from print3 import print3
>>> print3(3.12, 'py', sap='@')
3.12 py
```

It may not make sense to print superfluous keywords, but we can detect them manually by using `dict.pop()` to delete fetched keywords, and checking if the dictionary is not empty when we're done. The version in [Example 18-7](#) does—it's equivalent to the original, but triggers a built-in exception with a `raise` statement when unexpected keyword arguments are sent by a call (this is partly preview: we'll study exceptions and `raise` in depth in [Part VII](#)).

Example 18-7. *print3_pops.py*

```
"Use keyword-collector arguments with deletion and defaults"
import sys

def print3(*args, **kargs):
    sep = kargs.pop('sep', ' ')
    end = kargs.pop('end', '\n')
    file = kargs.pop('file', sys.stdout)
    if kargs: raise TypeError(f'extra keywords: {kargs}')
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

Notice that this file's name uses an *underscore* instead of a dash: because it's to be imported, its name must follow the rules for variables of [Chapter 11](#). This version works as before, but it now catches extra-neous keyword arguments:

```
>>> from print3_pops import print3
>>> print3(3.12, 'py', sep='@')
3.12@py
>>> print3(3.12, 'py', sap='@')
TypeError: extra keywords: {'sap': '@'}
```

It's OK to reimport the same `print3` name from a different file here: this simply replaces the prior version just like reassigning any other variable (more on this later in this book). That being coded, this example could also use keyword-only arguments to *automatically* validate configuration arguments, as the final variant in [Example 18-8](#) illustrates.

Example 18-8. *print3_kwonly.py*

```
"Use keyword-only arguments to emulate print"
import sys

def print3(*args, sep=' ', end='\n', file=sys.stdout):
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

This version works the same way for valid calls, but catches invalid keywords with keyword-only arguments instead of manual code, and is a prime example of how keyword-only arguments can address coding needs:

```
>>> from print3_kwonly import print3
>>> print3(3.12, 'py', sep='@')
3.12@py
>>> print3(3.12, 'py', sap='@')
TypeError: print3() got an unexpected keyword argument 'sap'
```

Given that this version of the function also requires *four fewer lines* of code than its predecessor, keyword-only arguments can simplify a specific category of functions that accept both arguments and options. A similar case can be made for positional-inly arguments versus manual code, but it's more

obscure, and this chapter has run out of space. For another example of keyword-only arguments at work, stay tuned for the iteration-timing case study in [Chapter 21](#).

And for more inspiration, also see the sidebar “[Why You Will Care: Customizing open](#)” on page 421. Much as we did there, our print emulator could be assigned to `builtins.print` to replace the built-in with our custom version everywhere in a program. There’s no reason to do that for a version that’s the *same*, of course, but this technique can be used to install a replacement printer that mods or extends the built-in (e.g., with logging).

Chapter Summary

In this chapter, we studied the second of two key concepts related to functions: the *arguments* used to send objects to a function. As we saw, arguments are passed to a function by assignment, which means by object reference (which really means by pointer), and are open to the usual side effects for shared mutable objects, desired or not. We also studied some advanced extensions that generalize argument matching, including default and keyword arguments, tools for collecting and unpacking arbitrarily many arguments, and keyword- and positional-only arguments. Finally, we explored a few larger examples that employed argument tools, and previewed module topics of this book’s next part.

The next chapter continues our look at functions with a grab bag of more advanced function-related ideas: function annotations, recursion, and more on `lambda` and functional tools such as `map` and `filter`. Many of these concepts stem from the fact that functions are normal objects in Python, and so support flexible processing tools and modes. Before diving into those topics, however, take this chapter’s quiz to review the argument ideas we’ve studied here.

Test Your Knowledge: Quiz

This quiz asks you to trace through examples of function definitions and calls to predict their outputs. Try to work out the answers on your own before resorting to cut and paste in a REPL:

1. What is the output of the following code, and why?

```
>>> def func(a, b=4, c=5):  
    print(a, b, c)  
  
>>> func(1, 2)
```

2. What is the output of this code, and why?

```
>>> def func(a, b, c=5):  
    print(a, b, c)  
  
>>> func(1, c=3, b=2)
```

3. How about this code: what is its output, and why?

```
>>> def func(a, *pargs):  
    print(a, pargs)  
  
>>> func(1, 2, 3)
```

4. What does this code print, and why?

```
>>> def func(a, **kargs):  
    print(a, kargs)  
  
>>> func(a=1, c=3, b=2)
```

5. What gets printed by this, and why?

```
>>> def func(a, b, c=3, d=4): print(a, b, c, d)

>>> func(1, *(5, 6))
```

6. One last time: what is the output of this code, and why?

```
>>> def func(a, b, c):
    a = 2; b[0] = 'x'; c['a'] = 'y'

>>> L=1; M=[1]; N={'a': 0}
>>> func(L, M, N)
>>> L, M, N
```

Test Your Knowledge: Answers

1. The output here is 1 2 5, because 1 and 2 are passed to a and b by position, and c is omitted in the call and hence defaults to 5.
2. The output this time is 1 2 3: 1 is passed to a by position, and b and c are passed 2 and 3 by name (the left-to-right order doesn't matter when keyword arguments are used like this).
3. This code prints 1 (2, 3), because 1 is passed to a and the *args collects the remaining positional arguments into a new tuple object. We can step through the extra positional arguments tuple with any iteration tool (e.g., for arg in args: ...).
4. This time the code prints 1 {'c': 3, 'b': 2}, because 1 is passed to a by name and the **kwargs collects the remaining keyword arguments into a dictionary. We could step through the extra keyword arguments dictionary by key with any iteration tool (e.g., for key in kwargs: ...). Note that the order of the dictionary's keys reflects the order in which keyword arguments are passed, in recent Pythons.
5. The output here is 1 5 6 4: the 1 matches a by position, 5 and 6 match b and c by * positional unpacking (6 overrides c's default), and d defaults to 4 because it was not passed a value.
6. This displays (1, ['x'], {'a': 'y'})—the first assignment in the function doesn't impact the caller, but the second two do because they change passed-in mutable objects in place.

Why You Will Care: Keyword Arguments

As you can probably tell, some argument-matching mode combos can be complex. They are also largely optional in your code; in fact, you can get by with just simple positional matching, and it's probably a good idea to do so when you're starting out. However, because many Python tools make good use of them, some general knowledge of these modes is important.

For example, keyword arguments play a key role in tkinter, the de facto standard GUI API for Python. We touch on tkinter only briefly at various points in this book, but in terms of its call patterns, keyword arguments set configuration options when GUI components are built. For instance, a call of the form:

```
from tkinter import Button
widget = Button(text='Press me', command=someFunction)
```

creates a new button and specifies its text displayed in the GUI and callback function run on a press, using the `text` and `command` keyword arguments. Since the number of configuration options for a widget can be large, keyword arguments let you pick and choose which to apply. Without them, you might have to either list all the possible options by position or hope for a judicious positional-argument defaults protocol that would handle every possible option arrangement.

Many built-in functions in Python expect us to use keywords for usage-mode options as well, which may or may not have defaults. As we learned in [Chapter 8](#), for instance, the `sorted` built-in:

```
sorted(iterable, key=None, reverse=False)
```

expects us to pass an iterable object to be sorted, but also allows us to pass in optional keyword arguments to specify both a value-transform function and a reversal flag, which default to `None` and `False`, respectively. Since we normally don't use these options, they may be omitted to apply defaults.

As we've also seen, the `dict`, `str.format`, and `print` calls accept keywords as well—other usages we had to introduce in earlier chapters because of their forward dependence on argument-passing modes we've finally studied here (alas, those who change Python already know Python!).

Function Odds and Ends

This chapter presents a medley of function-related topics: recursive functions; function attributes, annotations, and decorations; and more on both the `lambda` expression and functional-programming tools such as `map` and `filter`. These are all somewhat advanced tools that, depending on your job description, you may not encounter on a regular basis. Because of their roles in some domains, though, a basic understanding can be useful. `lambda`, for instance, makes regular appearances in GUIs, and functional programming techniques have grown common in Python code.

Some of the art of using functions lies in the *interfaces* between them, so we will also explore some general function design principles here. The next chapter continues the advanced themes here with an exploration of generator functions and expressions and a revival of list comprehensions in the context of the functional tools we will study here.

Function Design Concepts

Now that we've studied function essentials in Python, let's open this chapter with some perspective. When you start using functions in earnest, you're faced with choices about how to glue components together—for instance, how to decompose a task into purposeful functions (known as *cohesion*), and how your functions should communicate (called *coupling*). You also need to take note of the *size* of your functions because it directly impacts code usability. Some of this falls into the category of structured analysis and design, but it applies to Python code as to any other.

We explored some ideas related to function and module coupling in [Chapter 17](#) when studying scopes, but here is a review of a few general guidelines for readers new to function design principles:

- **Coupling: use arguments for inputs and return for outputs.** Generally, you should strive to make a function independent of the world outside of it. Arguments and `return` statements are often the best ways to isolate external dependencies to a small number of well-known places in your code.
- **Coupling: use global variables only when truly necessary.** As we've seen, global variables (i.e., names in the enclosing module) are usually a poor way for functions to communicate. They can create dependencies and timing issues that make programs difficult to debug, change, and reuse.

- **Coupling: don't change mutable arguments unless the caller expects it.** As we've also seen, functions can change parts of passed-in mutable objects, but as with global variables, this creates a tight coupling between the caller and callee, which can make a function too specific and brittle.
- **Cohesion: each function should have a single, unified purpose.** When designed well, each of your functions should do one thing—something you can summarize in a simple declarative sentence. If that sentence is very broad (e.g., “this function implements my whole program”) or contains lots of conjunctions (e.g., “this function gives employee raises *and* submits a pizza order”), you might want to think about splitting it into separate and simpler functions. Otherwise, there is no way to reuse the code of the individual steps embedded in the function.
- **Size: each function should be relatively small.** This naturally follows from the preceding goal, but if your functions start spanning multiple pages on your display, it's probably time to split them. Especially given that Python code is so concise to begin with, a long or deeply nested function is often a symptom of design problems. Keep it simple, and keep it short.
- **Coupling: avoid changing variables in another module file directly.** We also introduced this concept in [Chapter 17](#), and we'll revisit it in the next part of the book when we focus on modules. For reference, though, remember that changing variables across file boundaries sets up a coupling between modules similar to how global variables couple functions—the modules become difficult to understand and reuse separately. Use accessor functions whenever possible, instead of direct assignment statements.

Figure 19-1 summarizes the ways functions can talk to the outside world; inputs may come from items on the left side, and results may be sent out in any of the forms on the right. Nonlocals might belong in this sketch too, but they're mostly a state-retention tool in the same category as other local variables. Despite the array of options, good function designs prefer to use only arguments for inputs and return statements for outputs, whenever possible.

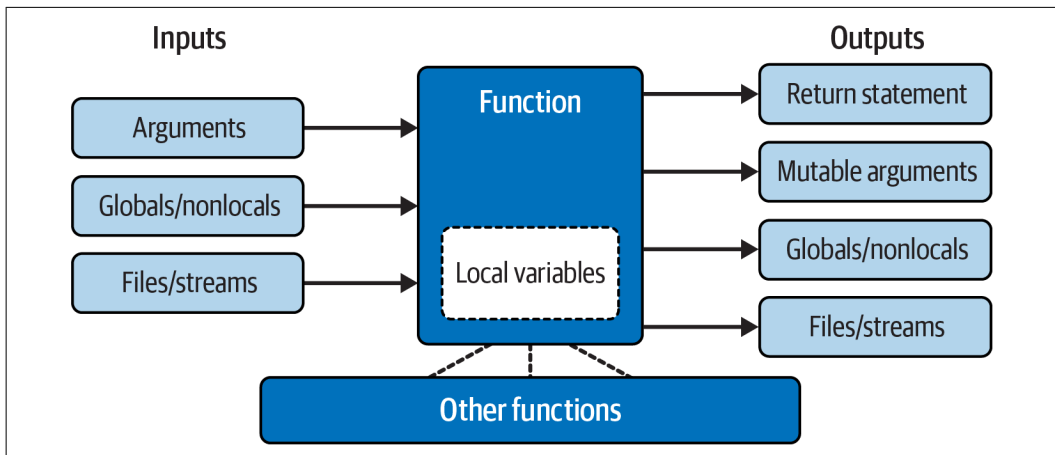


Figure 19-1. Function execution environment

Of course, there are plenty of exceptions to the preceding design rules, including some related to Python's OOP support. As you'll see in [Part VI](#), Python classes *depend* on changing a passed-in mutable object—class functions set attributes of an automatically passed-in argument called `self` to change per-object state information (e.g., `self.edition=6`). Moreover, if classes are not used, global variables are a

straightforward way for functions in modules to retain single-copy state between calls. Side effects are usually dangerous only if they're unexpected.

In general though, you should strive to minimize external dependencies in functions and other program components. The more *self-contained* a function is, the easier it will be to understand, reuse, and modify. Making code as freestanding as possible is especially important when functions go multilevel with recursion, per the next section.

Recursive Functions

We mentioned recursion in relation to comparisons of core types in [Chapter 9](#). While discussing scope rules near the start of [Chapter 17](#), we also briefly noted that Python supports *recursive functions*—functions that call themselves either directly or indirectly in order to loop. In this section, we'll explore what this looks like in our functions' code.

Recursion is a somewhat advanced topic, and it's relatively uncommon to see in Python, partly because Python's procedural shed includes simpler looping tools. Still, it's a useful technique to know about, as it allows programs to traverse structures that have arbitrary and unpredictable shapes and depths—planning travel routes, analyzing language, and crawling links on the web, for example. Recursion is even an alternative to simple loops and iterations, though not necessarily the simplest or most efficient one.

Summation with Recursion

Let's turn to some examples. To sum a list (or other sequence) of numbers, we can either use the built-in `sum` function or write a more custom version of our own. [Example 19-1](#) shows what a custom summing function might look like when coded with recursion.

Example 19-1. mysum.py

```
def mysum(L):
    if not L:
        return 0
    else:
        return L[0] + mysum(L[1:])    # Call myself recursively
```

To use, either add self-test code to the bottom of this file and run it as a script, or import it as a module and test at the REPL (again, the file may need to be in the folder where you're working either way, per [Chapter 3](#)). With the latter:

```
>>> from mysum import mysum    # Import file as a module in a REPL
>>> mysum([1, 2, 3, 4, 5])      # Sum all the numbers in any sequence
15
```

At each level, this function calls itself recursively to compute the sum of the *rest* of the list, which is later added to the item at the *front*. This recursive loop ends and zero is returned when the list becomes empty. When using recursion like this, each open level of call to the function has its own copy of the function's local scope on the runtime call stack. Here, that means `L` is different in each level, so each remembers its own segment of the list.

If this is difficult to understand (and it often is for new programmers), try adding a `print` of `L` to the function and run it again, to trace the current list at each call level; here's the required mod pasted at the REPL for variety:

```

>>> def mysum(L):
    print(L)                # Trace recursive levels
    if not L:                # L shorter at each level
        return 0
    else:
        return L[0] + mysum(L[1:])

>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15

```

As you can see, the list to be summed grows smaller at each recursive level, until it becomes empty—the termination of the recursive loop. The sum is then computed as the recursive calls unwind on returns.

Coding Alternatives

Interestingly, we can use Python’s if/else ternary expression (described in [Chapter 12](#)) to save some code real estate here. We can also generalize for any summable type (which is easier if we assume at least one item in the input) and use extended-unpacking assignment to make the first/rest unpacking simpler (as covered in [Chapter 11](#)). [Example 19-2](#) collects all three of these mods, ready to be run in a file or pasted into a REPL.

Example 19-2. *mysum_alts.py*

```

def mysum(L):
    return 0 if not L else L[0] + mysum(L[1:])    # Use ternary expression

def mysum(L):
    return L[0] if len(L) == 1 else L[0] + mysum(L[1:]) # Any type, assume one+

def mysum(L):
    first, *rest = L
    return first if not rest else first + mysum(rest)  # Use extended unpacking

```

When tested individually, all three of these alternatives handle numeric summation the same as the original:

```

>>> mysum([1, 2, 3, 4, 5])
15

```

Uniquely, the latter two fail for empties (e.g., `mysum([])`), but handle sequences of *any* object type that supports `+`, not just numbers (for strings, the effect is similar to `' '.join(L)`):

```

>>> mysum(('h', 'a', 'c', 'k'))    # The last two fail on mysum([])
'hack'
>>> mysum(['hack', 'app', 'code'])  # But they support nonnumeric types
'hackappcode'

```

Run some tests on your own for more insight. If you study these three variants, you’ll also find that:

- The latter two work on a single *string* argument (e.g., `mysum('hack')`), because strings are sequences of one-character strings (though this use case isn’t very useful: you get back the same string).

- The third variant also works on arbitrary *iterables*, including open input files (`mysum(open(name))`), but the others' indexing generally fails on nonsequences (see [Chapter 14](#) for extended-unpacking demos).

You may also notice that the third variant's unpacking assignment is similar to a `*` collector in a function header, and it's tempting to recode it as such. This won't quite work, though, because it would expect *individual* arguments, not a single iterable—unless we *also* star both the top-level input and recursive call. Here's the end result, though by summing discrete arguments, it solves a different problem than both the prior versions and built-in `sum`:

```
>>> def mysum(first, *rest):
    return first if not rest else first + mysum(*rest)

>>> mysum(*[1, 2, 3, 4, 5])
15
>>> mysum(*'hack')
'hack'
```

Finally, bear in mind that recursion can be either *direct*, as in the examples so far, or *indirect*, as in the following—a function that calls another function, which calls back to its caller. The net effect is the same, though there are two function calls at each level instead of one:

```
>>> def mysum(L):
    if not L: return 0
    return nonempty(L)                # Call a function that calls me

>>> def nonempty(L):
    return L[0] + mysum(L[1:])        # Indirectly recursive

>>> mysum([1.1, 2.2, 3.3, 4.4])
11.0
```

Loop Statements Versus Recursion

Though recursion works for summing in the prior sections' examples, it's probably overkill in this context. In fact, recursion is not used nearly as often in Python as in more esoteric languages like Prolog or Lisp, because Python emphasizes simpler procedural statements like loops, which are usually more natural. The `while`, for example, often makes things more concrete, and it doesn't require that a function be defined to allow recursive calls:

```
>>> L = [1, 2, 3, 4, 5]
>>> tot = 0
>>> while L:
    tot += L[0]
    L = L[1:]

>>> tot
15
```

Better yet, for loops iterate for us automatically, making recursion largely extraneous in many cases (and, in all likelihood, less efficient in terms of memory space and execution time):

```
>>> L = [1, 2, 3, 4, 5]
>>> tot = 0
>>> for x in L: tot += x

>>> tot
15
```

With looping statements, we don't require a fresh copy of a local scope on the call stack for each iteration, and we avoid the speed costs associated with function calls in general. (Stay tuned for [Chapter 21](#)'s timer case study for ways to compare the execution times of alternatives like these.)

Handling Arbitrary Structures

On the other hand, recursion—or equivalent and explicit stack-based algorithms we'll explore shortly—can be *required* to traverse arbitrarily shaped structures. As a simple example of recursion's role in this context, consider the task of computing the sum of all the numbers in a nested sublists structure like this:

```
[1, [2, [3, 4], 5], 6, [7, 8]]           # Arbitrarily nested sublists
```

Neither our prior summers nor simple looping statements will work here because this is not a linear iteration. Nested looping statements do not suffice either—because the sublists may be nested to arbitrary *depth* and in an arbitrary *shape*, there's no way to know how many nested loops to code to handle all cases. Instead, the function in [Example 19-3](#) accommodates such general nesting by using recursion to visit sublists along the way.

Example 19-3. *sumtree.py*

```
def sumtree(L, trace=False):
    tot = 0
    for x in L:
        # For each item at this level
        if not isinstance(x, list):
            tot += x
            # Add numbers directly
            if trace: print(x, end=', ')
        else:
            tot += sumtree(x, trace)
            # Recur for sublists
    return tot
```

In this file's function, each recursive level runs a for loop to add numbers, or recur into sublists to open new levels. Recall from [Chapter 9](#) that `isinstance` compares object types; it's used here to detect nested sublists.

This code is also instrumented to trace items as they are added to the total: if `trace` is passed a true value, you can see how the object is scanned left to right. Because it also steps down into sublists with recursion along the way, though, the traversal is really both horizontal and vertical:

```
>>> from sumtree import sumtree
>>> sumtree([1, [2, [3, 4], 5], 6, [7, 8]])
36
>>> sumtree([1, [2, [3, 4], 5], 6, [7, 8]], trace=True)
1, 2, 3, 4, 5, 6, 7, 8, 36
```

Testing with a separate script

At this point, we could test other cases by typing them interactively or by adding code to the bottom of the file, but you're probably starting to see that this can be a bit limiting. Instead, the script in [Example 19-4](#) makes the process automatic and easily repeatable. As a bonus, it can be used for other summers we'll code in a moment.

Example 19-4. *sumtree_tester.py*

```
tests = (  
[1, [2, [3, 4], 5], 6, [7, 8]],      # Mixed nesting => 36  
[1, [2, [3, [4, [5]]]],              # Right-heavy nesting => 15  
[[[[[1], 2], 3], 4], 5])            # Left-heavy nesting => 15  
  
def tester(sumtree, trace=True):  
    for test in tests:  
        print(sumtree(test, trace))
```

To use this tester, simply import both `summer` and `tester`, and pass the former to the latter. When run, our `summer` prints numbers as added with the final sum at the end, for each of three canned tests:

```
>>> from sumtree import sumtree          # Get the summer  
>>> from sumtree_tester import tester    # Get the tester  
>>> tester(sumtree)                      # Run the tester on the summer  
1, 2, 3, 4, 5, 6, 7, 8, 36  
1, 2, 3, 4, 5, 15  
1, 2, 3, 4, 5, 15
```

Within `tester`, `sumtree` refers to the `summer` function passed into it. Again, because functions are objects, passing them around this way is natural, and makes code flexible.

Recursion versus queues and stacks

It sometimes helps recursion newcomers to understand that internally, Python implements recursion by pushing information on a *call stack* at each recursive call, so it remembers where it must return and continue later. In fact, it's generally possible to implement recursive-style procedures *without* recursive calls, by using an explicit stack or queue of your own to keep track of remaining steps.

For instance, [Example 19-5](#) computes the same sums as the prior example, but uses an explicit list to schedule when it will visit items in the subject, instead of issuing recursive calls. The item at the front of the list is always the next to be processed and summed.

Example 19-5. *sumtree_queue.py*

```
def sumtree(L, trace=False):              # Breadth-first, explicit queue  
    tot = 0  
    items = list(L)                       # Start with copy of top level  
    while items:  
        front = items.pop(0)              # Fetch/delete front item  
        if not isinstance(front, list):  
            tot += front                   # Add numbers directly  
            if trace: print(front, end=', ')  
        else:  
            items.extend(front)            # <== Append all in nested list  
    return tot
```

Technically, this code traverses the list in *breadth-first* fashion (across before down), because it adds nested lists' contents to the *end* of the list—forming a FIFO (first-in-first-out) *queue*. The net effect sums by horizontal levels. To test, we can either import and use the new `summer` directly, or route it to the [Example 19-4](#) tester to be exercised automatically with tracing:

```

>>> from sumtree_queue import sumtree           # Get the new summer
>>> sumtree([1, [2, [3, 4], 5], 6, [7, 8]])
36
>>> from sumtree_tester import tester           # Unless already imported
>>> tester(sumtree)                             # Run tester on _this_ summer
1, 6, 2, 5, 7, 8, 3, 4, 36
1, 2, 3, 4, 5, 15
5, 4, 3, 2, 1, 15

```

Fine points: we don't have to reimport the tester again if it's already been imported in this session, and importing the same-named summer just works—the new summer's filename makes it unique, and sumtree is always the latest version imported if you import more than one, because imports *assign* names (see [Chapter 18](#)'s print emulators for another example of this pattern at work, and watch for more on imports in this book's next part).

More importantly, notice how the order in which numbers are visited here is different than in the original recursive-call version, due to the breadth-first queue. Trace through the tester's tests to see how this pans out.

If we instead want to emulate the traversal of the recursive-call version more closely, we can change this code to perform *depth-first* traversal (down before across) simply by adding the contents of nested lists to the *front* of the list—forming a last-in-first-out (LIFO) *stack*. [Example 19-6](#) makes the required mods, but the only way it differs from the breadth-first version is the line that adds to the front instead of the end, marked with `<==` in a comment.

Example 19-6. *sumtree_stack.py*

```

def sumtree(L, trace=False):                     # Depth-first, explicit stack
    tot = 0
    items = list(L)                             # Start with copy of top level
    while items:
        front = items.pop(0)                     # Fetch/delete front item
        if not isinstance(front, list):
            tot += front                         # Add numbers directly
            if trace: print(front, end=', ')
        else:
            items[:0] = front                   # <== Prepend all in nested list
    return tot

```

As before, we can use this function directly, or pass it to the same tester; its file makes it distinct, and its name refers to the latest import. When run, this summer visits numbers in the same order as the recursive-calls original, but manages the traversal with an explicit stack instead of recursion:

```

>>> from sumtree_stack import sumtree           # Same name, different file
>>> sumtree([1, [2, [3, 4], 5], 6, [7, 8]])
36
>>> from sumtree_tester import tester           # Optional if already imported
>>> tester(sumtree)
1, 2, 3, 4, 5, 6, 7, 8, 36
1, 2, 3, 4, 5, 15
1, 2, 3, 4, 5, 15

```

For more on the last two examples (plus another breadth-first coding variant omitted here), see file [sumtree_etc.py](#) in the book's examples package. It adds additional tracing so you can watch it walk structures in more detail.

In general, though, once you get the hang of recursive calls, they may be more natural than the explicit scheduling lists they automate, and are generally preferred unless you need to traverse structures in specialized ways. Some programs, for example, perform a *best-first* search that requires an explicit search queue ordered by relevance or other criteria. If you think of a web crawler that scores sites visited by content, the applications may start to become clearer.

Cycles, paths, and stack limits

As is, these programs suffice as demos, but larger recursive applications can sometimes require a bit more infrastructure than shown here: they may need to avoid cycles or repeats, record paths taken for later use, and expand stack space when using recursive calls instead of explicit queues or stacks.

For instance, neither the recursive-call nor the explicit queue/stack examples in this section do anything about avoiding *cycles*—visiting a location already visited. That’s not required here, because we’re traversing strictly hierarchical trees of list objects. If data can be a cyclic graph, though, both these schemes will fail: the recursive-call scheme will fall into an infinite recursive loop (and may run out of call-stack space), and the others will fall into simple infinite loops, re-adding the same items to their lists (and may or may not run out of general memory). In fact, it’s easy to demo the perils by creating a cyclic object with the strange code we met in an exercise at the end of [Chapter 3](#):

```
>>> L = [1, 2]
>>> L.append(L)      # Make a cyclic object: L references itself
>>> L
[1, 2, [...]]

>>> from sumtree import sumtree
>>> sumtree(L)
RecursionError: maximum recursion depth exceeded

>>> from sumtree_queue import sumtree
>>> sumtree(L)
...hang or crash, and ditto for stack...
```

Some programs also need to avoid repeated processing for a state reached more than once, even if that wouldn’t lead to a loop. To do better, a recursive-call traversal might make and pass along a mutable set, dictionary, or list of states visited so far and check for repeats as it goes. We will use this scheme in later recursive examples in this book:

```
if state not in visited:
    visited.add(state)      # x.add(state), x[state]=True, or x.append(state)
    ...proceed...
```

Nonrecursive alternatives might similarly avoid adding states already visited with code like the following. Subtly, object cycles may require `is` (not `in`), and simply checking for duplicates already on the `items` list would avoid scheduling a state twice but would not prevent revisiting a state visited earlier and hence removed from that list:

```
visited.add(front)
...proceed...
items.extend([x for x in front if x not in visited])
```

This model doesn’t quite apply to this section’s use case that simply adds numbers in lists, but other applications will generally be able to identify repeated states—a URL of a previously visited web page, for instance. In fact, we’ll use such techniques to avoid cycles and repeats in the later examples listed in the next section.

Some programs may also need to record complete *paths* for each state followed so they can report solutions when finished. In such cases, each item in the nonrecursive scheme's stack or queue may be a full path list that suffices for a record of states visited, and contains the next item to explore at either end.

Also note that standard Python limits the *depth* of its runtime call stack—crucial to recursive-call programs—to trap infinite recursion errors. To expand it for deeper journeys, use the `sys` module:

```
>>> sys.getrecursionlimit()      # 1000 calls deep default
1000
>>> sys.setrecursionlimit(10000)  # Allow deeper nesting
>>> help(sys.setrecursionlimit)    # Read more about it
```

The maximum allowed setting can vary per platform. This isn't required for programs that use stacks or queues to avoid recursive calls and gain more control over the traversal process (though they also won't catch infinite loops).

More recursion examples

Although this section's example is artificial, it is representative of a larger class of programs; inheritance trees and module import chains, for example, can exhibit similarly general structures, and computing tools such as permutations can require arbitrarily many nested loops. In fact, we'll use recursion again in such roles later in this book:

- In [Chapter 20's *permute.py*](#), to shuffle arbitrary sequences
- In [Chapter 25's *reloadall.py*](#), to traverse import chains
- In [Chapter 29's *classtree.py*](#), to traverse class inheritance trees
- In [Chapter 31's *lister.py*](#), to traverse class inheritance trees again
- In [Appendix B, “Solutions to End-of-Part Exercises”](#) at the end of this part of the book: count-downs and factorials

The second and third of these will also detect states already visited to avoid cycles and repeats. Although simple loops should generally be preferred to recursion for linear iterations on the grounds of simplicity and efficiency, you'll find that recursion is essential in scenarios like those in these later examples.

Moreover, you sometimes need to be aware of the potential of *unintended* recursion in your programs. As you'll also see later in the book, some operator-overloading methods in classes such as `__setattr__` and `__getattr__` and even `__repr__` have the potential to recursively loop if used incorrectly. Recursion is a powerful tool, but it tends to be best when both understood and expected!

Function Tools: Attributes, Annotations, Etc.

Let's move on to a category of tools that may seem less ethereal than recursion to some earthlings. As we've seen in this part of the book, functions in Python are much more than code-generation specifications for a compiler—they are full-blown *objects*, stored in pieces of memory all their own. As such, they can be freely passed around a program and called indirectly. They also support operations that have little to do with calls at all, including attributes and annotation. We've sampled some of these topics earlier, but this section provides expanded coverage.

The First-Class Object Model

Because Python functions are objects, you can write programs that process them generically. Function objects may be reassigned to other names, passed to other functions, embedded in data structures, returned from one function to another, and more, as if they were simple numbers or strings. Function objects happen to support a special operation—they can be *called* by listing arguments in parentheses—but they belong to the same general category as other objects.

As we’ve seen, this is usually called a *first-class object model*; it’s ubiquitous in Python, and a necessary part of *functional programming*. We’ll explore this programming mode more fully in this and the next chapter; because its motif is founded on the notion of applying functions, it treats functions as a kind of data.

We’ve explored some generic use cases for functions in earlier examples, but a quick review helps to underscore the model. For example, there’s really nothing special about the name used in a `def` statement: it’s just a variable assigned in the current scope, as if it had appeared on the left of an `=` sign. Because the function name is simply a reference to an object after a `def` runs, you can *reassign* that object to other names freely and call it through any reference:

```
>>> def exclaim(message):           # Name exclaim assigned to function object
    print(message + '!')

>>> exclaim('Direct call')          # Call object through original name
Direct call!

>>> x = exclaim                     # Now x references the function too
>>> x('Indirect call')              # Call object through name x by adding ()
Indirect call!
```

And because arguments are passed by assigning objects, it’s just as easy to *pass* functions to other functions as arguments. The callee may then call the passed-in function just by adding arguments in parentheses (see the earlier summer tester in [Example 19-4](#) for another example of this pattern):

```
>>> def generic(func, arg):         # Call the passed-in object by adding ()
    func(arg)

>>> generic(exclaim, 'Argument call') # Pass the function to another function
Argument call!
```

You can even stuff function objects into data structures, as though they were integers or strings. The following, for example, *embeds* the function twice in a list of tuples, as a sort of actions table. Because Python compound types like these can contain any sort of object, there’s no special case here, either ([Example 18-2](#) used similar code):

```
>>> schedule = [ (exclaim, 'Hack'), (exclaim, 'Code') ]
>>> for (func, arg) in schedule:
    func(arg)                       # Call functions embedded in containers

Hack!
Code!
```

This code simply steps through the `schedule` list, calling the `exclaim` function with one argument each time through. As we saw in [Chapter 17](#)’s examples, functions can also be created and *returned* for use elsewhere—the *closure* created in this mode also retains state from the enclosing scope:

```
>>> def implore(verb):                # Make a function but don't call it
    def exclaim(noun):
        print(f'{verb} {noun}!')
    return exclaim

>>> I = implore('Hack')              # Label in enclosing scope is retained
>>> I('Code')                        # Call the function that make returned
Hack Code!
>>> I('App')
Hack App!
```

Python's first-class object model and lack of type constraints make for a very flexible programming language.

Function Introspection

In fact, functions are more flexible than you might expect. Because they are objects, we can also process functions with normal object tools. For instance, by now we know that once we make a function we can call it as usual:

```
>>> def func(a):
    b = 'Hack'
    return b * a

>>> func(8)
'HackHackHackHackHackHackHackHack'
```

But the call expression is just one of a set of operations defined to work on function objects. For instance, we can also inspect their attributes generically:

```
>>> func.__name__
'func'
>>> dir(func)
['__annotations__', '__builtins__', '__call__', '__class__', '__closure__',
...more omitted: 38 total...
['__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__type_params__']
```

Introspection—internals access—tools like this allow us to explore implementation details. For example, functions have attached *code objects*, which provide details on aspects such as the functions' local variables and arguments:

```
>>> func.__code__
<code object func at 0x103ef3910, file "<stdin>", line 1>

>>> dir(func.__code__)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
...more omitted: 48 total...
'co_posonlyargcount', 'co_qualname', 'co_stacksize', 'co_varnames', 'replace']

>>> func.__code__.co_varnames
('a', 'b')
>>> func.__code__.co_argcount
1
```

Tool writers can make use of such information to manage functions—in fact, we will too in [Chapter 39](#), to implement validation of function arguments with the decorators introduced ahead. Whether you code or use such tools, object introspection boosts function utility.

Function Attributes

Nor are function objects limited to the system-defined attributes of the prior section: it's also possible to attach arbitrary *user-defined* attributes to them. This topic was introduced in [Chapter 17](#), but this section expands on it with additional context and examples. As usual in Python, function attributes are created by simple assignments:

```
>>> def func(): pass
>>> func
<function func at 0x1043771a0>
>>> func.count = 0
>>> func.count += 1
>>> func.count
1
>>> func.handles = 'Button-Press'
>>> func.handles
'Button-Press'
>>> dir(func)
...most double-underscore names omitted...
['__str__', '__subclasshook__', '__type_params__', 'count', 'handles']
```

Python's own implementation-related data stored on functions follows naming conventions that prevent them from clashing with the more arbitrary attribute names you might assign yourself. Specifically, all function internals' names have leading and trailing double underscores (“`__X__`”):

```
>>> len(dir(func))
40
>>> [x for x in dir(func) if not x.startswith('__')]
['count', 'handles']
```

If you're careful not to name attributes the same way as Python, you can safely use the function's namespace as though it were your own namespace or scope. Naturally, all of this works the same for functions made with `lambda`:

```
>>> F = lambda: None
>>> len(dir(F))
38
>>> F.book = 'LP6E'
>>> F.book
'LP6E'
```

As covered in [Chapter 17](#), such attributes can be used to attach *state information* to function objects directly, instead of using other techniques such as globals, nonlocals, and classes. Unlike nonlocals, such attributes are accessible anywhere the function itself is, even from outside its code.

In a sense, this is also a way to emulate “static locals” in other languages—variables whose names are local to a function, but whose values are retained after a function exits. Attributes are related to objects instead of scopes (and must be referenced through the function name within its code), but the net effect is similar.

Moreover, as also explored in [Chapter 17](#), when attributes are attached to functions generated by other *factory* functions, they also support multiple copy, writeable, and *per-call* state retention, as an alternative to closures and class-instance attributes. This makes function attributes a broadly useful tool—like the topics of the next section.

Function Annotations and Decorations

For tools roles, functions also support attached *annotations*—arbitrary user-defined info about a function's arguments and result that augment the function. Python provides syntax for coding annotations, but it doesn't do anything with them itself—annotations are completely optional, don't impact function behavior in any way, and when present are simply attached to the function object's `__annotations__` attribute for use by other tools.

While not of general interest to most application programmers, third-party tools and libraries might use annotations in the context of enhanced error checking, or API directives. Type hinting, discussed in [Chapter 6](#), is also based on annotations, but is optional and unused, and doesn't preclude other roles for annotations today (more on this ahead).

We studied the formal rules for arguments in function definitions in the preceding chapter. Annotations don't modify these rules but simply extend their syntax to allow extra expressions to be associated with named arguments and function results. Consider the following nonannotated function, coded with three arguments and a returned result:

```
>>> def func(a, b, c):
        return a + b + c

>>> func(1, 2, 3)
6
```

Syntactically, function annotations are coded in `def` header lines (only), as arbitrary expressions associated with arguments and return values. For arguments, they appear after a colon immediately following the argument's name; for return values, they are written after a `->` following the arguments list's closing parenthesis. The following code, for example, annotates all three of the prior function's arguments, as well as its return value:

```
>>> def func(a: 'hack', b: (1, 10), c: float) -> int:
        return a + b + c

>>> func(1, 2, 3)
6
```

Calls to an annotated function work as usual, but when annotations are present Python collects them in a *dictionary* and attaches it to the function object itself as its `__annotations__`. In this, argument names become keys; the return value annotation is stored under key `return` if coded (which suffices because this reserved word can't be used as an argument name); and the values of annotation keys are assigned to the results of the annotation expressions:

```
>>> func.__annotations__
{'a': 'hack', 'b': (1, 10), 'c': <class 'float'>, 'return': <class 'int'>}
```

Because they are just Python objects attached to a Python object, annotations are straightforward to process. The following annotates just two of three arguments and steps through the attached annotations generically:

```
>>> def func(a: 'python', b, c: 3.12):
        return a + b + c

>>> func(1, 2, 3)
6
>>> func.__annotations__
{'a': 'python', 'c': 3.12}

>>> for arg in func.__annotations__:
```

```

    print(arg, '=>', func.__annotations__[arg])

a => python
c => 3.12

```

There are two fine points to note here. First, you can still use *defaults* for arguments if you code annotations—the annotation (and its `:` character) appears *before* the default (and its `=` character). In the following, for example, `a: 'hack' = 4` means that argument `a` defaults to 4 and is annotated with the string `'hack'`:

```

>>> def func(a: 'hack' = 4, b: (1, 10) = 5, c: float = 6) -> int:
    return a + b + c

>>> func(1, 2, 3)                # No defaults used
6
>>> func()                       # a=4 + b=5 + c=6 (all defaults)
15
>>> func(1, c=10)                # 1 + b=5 + 10 (keywords work normally)
16
>>> func.__annotations__
{'a': 'hack', 'b': (1, 10), 'c': <class 'float'>, 'return': <class 'int'>}

```

Second, note that the *blank spaces* in the prior example are all optional—you can use spaces between components in function headers or not, but omitting them might degrade your code’s readability to some observers (and probably improve it to others):

```

>>> def func(a:'hack'=4, b:(1,10)=5, c:float=6)->int:
    return a + b + c

>>> func(1, 2)                  # 1 + 2 + c=6
9
>>> func.__annotations__
{'a': 'hack', 'b': (1, 10), 'c': <class 'float'>, 'return': <class 'int'>}

```

While annotations are optional and uncommon, they may be used to specify constraints for types or values, and larger APIs might use this feature as a way to register function interface information. In fact, you’ll see a potential application in [Chapter 39](#), when we code annotations as an alternative to *function decorator arguments*—a more general concept in which augmentation info is coded *outside* the function header and so is not limited to a single role.

Function decorators alternative: Preview

Because we’re going to devote an entire chapter to decorators, we’ll omit most of their story here. But as a very brief preview, decorators are simply functions that augment other functions. They are applied to another function’s `def` with an `@` prefix that rebinds the other function’s name to the result of passing it to the decorator. This syntax:

```

@decorator
def func(args): ...           # Decorated function def

```

is automatically morphed into the following equivalent, where `decorator` is a one-argument callable object (or an expression that returns one), which returns a callable object having arguments compatible with `func` (or `func` itself):

```

def func(args): ...
func = decorator(func)       # Rebind func to decorator's result

```

Decorators can use this hook to wrap another function in an extra layer of code for nearly arbitrary purposes, as in the following code that adds a message when a decorated function is called:

```
def echo(F):
    def proxy(*args):
        print('calling', F.__name__)    # Add actions here
        return F(*args)                # Run decorated function
    return proxy

@echo
def func(x, y):
    print('I am running...', x, y)     # func is run by the proxy closure

>>> func(1, 2)                        # Runs a proxy, which runs func
calling func
I am running... 1 2
```

As you’ll learn later, decorators can also take *arguments* (e.g., `@echo(args)`) whose utility can overlap with annotations—argument info can be sent to the decorator instead of being embedded in headers as annotations. Because this is an advanced tool that can also be applied to classes, we’ll pause this thread until Chapters 32 and 39.

Compared to annotations, though, decorators don’t complicate function headers themselves with extra syntax, and more naturally support multiple roles. Annotations may make functions difficult to read, and generally can have just one role—one that’s hijacked by the optional *type hinting* of Chapter 6 in programs that choose to employ it.

In fact, type hinting advocates have tried to deprecate all other roles for annotations. While these divisive (and perhaps even rude) efforts have thankfully failed to date, decorators face no such challenge, and may be a safer bet going forward. Today, though, both annotations and decorations are tools whose roles are limited only by your imagination.

Finally, note that annotations and decorations work in `def` statements—but not in `lambda` expressions, whose syntax by design limits the functions they can define. Coincidentally, this brings us to this pot-pourri’s next topic.

Anonymous Functions: `lambda`

We first met the `lambda` expression back in Chapter 16 and have used it in a handful of examples since then. This section reviews the basics and takes a deeper second look, to both reinforce and expand on this topic.

As we’ve seen, besides the `def` statement, Python provides an expression that creates function objects. Because of its similarity to a tool in other languages, it’s called `lambda`. Like `def`, this expression creates a function to be called later, but it returns the function instead of assigning it to a name. This is why `lambdas` are sometimes known as *anonymous* functions. In practice, they are used to *inline* function definitions, or *defer* execution of code.



What’s in a name?: The `lambda` tends to intimidate people, largely due to the name “`lambda`” itself—a name that comes from the Lisp language, which got it from *lambda calculus*, which is a form of symbolic logic. Obscure mathematical heritage aside, though, `lambda` in Python is really just a word that introduces an expression syntactically, and its expression is simply an alternative way to code a function—albeit without statements, decorators, or annotations.

lambda Basics

As a refresher, the `lambda`'s general form is the keyword `lambda`, followed by zero or more arguments (just like the arguments you enclose in parentheses in a `def` header, sans annotations), followed by an expression after a colon:

```
lambda argument1, argument2,... argumentN : expression-using-arguments
```

Parentheses are not allowed around `lambda` arguments and are generally optional around the entire `lambda` itself, though they're required in some contexts and may boost clarity in others. Functions returned by `lambda` work the same as those assigned by `def`, but `lambda` differs in ways that make it useful in specialized roles:

- **lambda is an expression, not a statement.** Because of this, a `lambda` can appear in places a `def` is not allowed by Python's syntax—inside a list literal or a function call's arguments, for example. With `def`, functions can be referenced by name in such locations, but must be created elsewhere. As an expression, `lambda` returns a value (a new function) that can be assigned a name, or used where the `lambda` appears.
- **lambda's body is a single expression, not a block of statements.** The `lambda`'s body is similar to what you'd put in a `def` body's return statement; you simply type the result as a naked expression, instead of explicitly returning it. Because it is limited to an expression, a `lambda` is less general than a `def`—you can only squeeze so much logic into a `lambda` body without full-blown statements. This is by design, to limit program nesting: `lambda` is designed for coding simple functions, and `def` handles larger tasks.

Apart from those distinctions, `defs` and `lambdas` do the same sort of work. For instance, by this point we should be pros at making a function with a `def` statement:

```
>>> def func(x, y, z): return x + y + z
>>> func(2, 3, 4)
9
```

But we can achieve the same effect with a `lambda` expression by explicitly assigning its result to a name through which you can later call the otherwise-anonymous function:

```
>>> func = lambda x, y, z: x + y + z
>>> func(2, 3, 4)
9
```

Here, `func` is manually assigned the function object the `lambda` expression creates; this is how `def` works, too, but its assignment is automatic. Defaults and other *argument-matching* syntax work in `lambda` too, just like in a `def`:

```
>>> x = (lambda a='hack', b='python', c='code': a + b + c)
>>> x('write')
'writepythoncode'
```

The code in a `lambda` body also follows the same *scope* lookup rules as code inside a `def`. `lambda` expressions introduce a new local scope much like a nested `def`, which automatically sees names in enclosing functions, the module, and the built-in scope—via the LEGB rule we studied in [Chapter 17](#):

```
>>> def editions(title):           # title in enclosing scope
    return (lambda e: title + ', ' + e) # Return a function object

>>> labeler = editions('Learning Python') # Make+save nested function
>>> labeler('6E')                     # '6E' passed to e in lambda
'Learning Python, 6E'
```

With those basic lambda “hows” in hand, the natural next question is “why,” taken up in the next section.

Why Use lambda?

Generally speaking, lambda is a sort of function *shorthand* that allows you to embed a function’s definition within the code that uses it. It is entirely optional—you can always use `def` instead, and *should* if your function requires the power of full statements that the lambda’s expression cannot provide. But lambda may be easier to use in scenarios where you just need to embed a small bit of executable code inline at the place where it is to be later used.

For instance, you’ll see later that callback handlers are frequently coded as inline lambda expressions embedded directly in a registration call’s arguments list, instead of being defined with a `def` elsewhere in a file and referenced by name (see the sidebar “[Why You Will Care: lambda Callbacks](#)” on page 480 for an example).

lambda is also commonly used to code *jump tables*, which are lists or dictionaries of actions to be performed on demand. For example:

```
L = [lambda x: x * 2,           # Inline function definition
     lambda x: x ** 2,
     lambda x: x // 2]          # A list of three callable functions

for f in L:
    print(f(5))                # Prints 10, 25, and 2

print(L[1](5))                 # Prints 25
```

The lambda expression works well when you need to stuff small pieces of executable code into places where statements like `def` are not allowed. The preceding code snippet, for example, builds up a list of three functions by embedding lambda expressions inside a list literal; a `def` won’t work inside a literal like this because it is a statement, not an expression. The equivalent `def` coding would require temporary function names (which might clash with others) and function definitions outside the context of intended use (which might be hundreds of lines away):

```
def f1(x): return x * 2
def f2(x): return x ** 2          # Define named functions
def f3(x): return x // 4         # Separate from their place of use

L = [f1, f2, f3]                 # Reference by name

for f in L:
    print(f(2))                  # Also prints 10, 25, and 2
```

Multiway branches: The finale

In fact, you can do the same sort of thing with dictionaries and other data structures in Python to build up more general sorts of action tables. Here’s another example to illustrate at the interactive prompt:

```
>>> key = 'loop'
>>> {'hack': lambda s: s.upper(),
    'code': lambda s: s.lower(),
    'loop': lambda s: f'{s * 4}!'}[key]('Py')
'PyPyPyPy!'
```

Here, when Python makes the temporary dictionary, each of the nested `lambdas` generates and leaves behind a function to be called later. Indexing by `key` fetches one of those functions, and parentheses force the fetched function to be called. When coded this way, a dictionary becomes a more general multiway branching tool than this book could fully reveal in [Chapter 12](#)’s coverage of `if` statements.

To make this work without `lambda`, you’d need to instead code three `def` statements somewhere else in your file, outside the dictionary in which the functions are to be used, and reference the functions by name:

```
>>> def f1(s): return s.upper()
>>> def f2(s): return s.lower()
>>> def f3(s): return f'{s * 4}!'

>>> key = 'loop'
>>> {'hack': f1, 'code': f2, 'loop': f3}[key]('Py')
'PyPyPyPy!'
```

This works, too, but your `defs` may be arbitrarily far away in your file, even if they are just little bits of code. The *code proximity* that `lambda` provides is especially useful for functions that will only be used in a single context—if the three functions here are not useful anywhere else, it makes sense to embed their definitions within the dictionary as `lambdas`. Moreover, the `def` form requires you to make up names for these little functions that, again, may clash with other names in this file (perhaps unlikely, but always possible).

You can use the `match` statement today with similar results, but it may require even more code than the `def` scheme, especially since you’d have to nest it within a `def` to support an argument like `s`; see [Chapter 12](#) for more info. `lambda` is also handy in function-call *argument lists* as a way to inline temporary function definitions not used anywhere else in your program; you’ll see examples of such other uses later in this chapter, when we study `map`.



The siren song of eval: In principle, you could skip the dispatch table in the preceding code if the function name is the same as its string lookup key—an `eval(key)(arg)` would kick off the call. While true in this case and sometimes useful, as we saw earlier (e.g., [Chapter 10](#)), `eval` is relatively slow (it must compile and run code) and insecure (you must trust the string’s source). More fundamentally, jump tables are generally subsumed by *polymorphic* method dispatch in Python: calling a method does the “right thing” based on the type of object that’s the subject of the call, no switching logic required. To see why, stay tuned for [Part VI](#).

How (Not) to Obfuscate Your Python Code

The fact that the body of a `lambda` has to be a single expression (not a series of statements) would seem to place severe limits on how much logic you can pack into a `lambda`. If you know what you’re doing, though, you can code most statements in Python as expression-based equivalents.

For example, if you want to *print* from the body of a `lambda` function, simply use `print` or `sys.stdout.write` (recall from [Chapter 11](#) that this is what `print` really does). And to execute *multiple* actions, code a sequence like a tuple, to evaluate nested multiple expressions from left to right (tuples require parentheses in this context):

```
>>> series = lambda a, b: (print(a.upper()), print(b.lower()))    # > 1 actions
>>> ignore = series('Hack', 'Code')
HACK
code
```

Similarly, to nest logic in a `lambda`, you can use the `if/else` ternary expression introduced in [Chapter 12](#), or the equivalent but trickier `and/or` combination also described there. As you learned earlier, the following statement:

```
if a:
    b
else:
    c
```

can be emulated by either of these equivalent expressions (the second is only roughly the same, but close enough):

```
b if a else c
((a and b) or c)
```

Because expressions like these can be placed inside a `lambda`, they may be used to implement *selection* logic within a `lambda` function (the `lambda` is parenthesized here only for variety and subjective clarity):

```
>>> lower = (lambda x, y: x if x < y else y)    # Ifs in lambda: ternary
>>> lower('bb', 'aa')
'aa'
>>> lower('aa', 'bb')
'aa'
```

Furthermore, if you need to perform *loops* within a `lambda`, you can also embed things like list comprehension expressions and `map` calls—tools we met in earlier chapters and will revisit in this and the next chapter:

```
>>> showall = lambda x: [print(y) for y in x]    # Loops in lambda: list comp
>>> t = showall(['lp5e', 'lp6e'])
lp5e
lp6e
>>> showall = lambda x: list(map(print, x))    # Loops in lambda: maps
>>> t = showall(['py3.3', 'py3.12'])
py3.3
py3.12
>>> showall = lambda x: print(*x, sep='', end='')    # Another option: *unpacking
```

And while it's limited by the local scope of the function that `lambda` makes, *assignment* is in scope (pun intended) for `lambda` expressions that use the `:=` named-assignment expression:

```
>>> namer = lambda x: (res := x + 1) + res    # Assignment in lambda: :=
>>> namer(2)
6
>>> res
NameError: name 'res' is not defined    # But it doesn't persist
```

There is a limit to emulating statements with expressions: you can't assign nonlocals, for instance, though tools like the `setattr` built-in, the `__dict__` of namespaces, and methods that change mutable

objects in place can sometimes stand in—and can quickly lead you deep into the heart of expression-convolution darkness.

But now that this book has shown you these tricks, it must also humbly implore you to use them only as a last resort. Without due care, they can yield unreadable (a.k.a. *obfuscated*) Python code, despite the language’s best intents. In general, simple is better than complex, explicit is better than implicit, and full statements are better than arcane expressions. That’s why `lambda` is limited to expressions. If you have larger logic to code, use `def`; `lambda` is for small pieces of inline code. On the other hand, you may find these techniques useful—when taken in moderation.

Scopes: lambdas Can Be Nested Too

One last `lambda` note: as we saw in [Chapter 17](#), `lambda` is also the main beneficiary of enclosing-function scope lookup—the *E* in the LEGB scope rule. As a review, in the following the `lambda` appears inside a `def`, its typical coding, and so can access the value that name `x` had in the enclosing function’s scope during its call:

```
>>> def action(x):
    return (lambda y: x + y)           # Make function, remember x

>>> act = action(99)
>>> act(2)                             # Call what action returned
101
```

What wasn’t illustrated in the prior discussion of nested function scopes is that a `lambda` also has access to the names in any enclosing `lambda`. This case is somewhat obscure, but imagine if we recoded the prior `def` with a `lambda`:

```
>>> action = (lambda x: (lambda y: x + y))   # lambda scopes nest too
>>> act = action(99)
>>> act(3)
102
>>> ((lambda x: (lambda y: x + y))(99))(4)   # Even if you don't save them
103
```

Here, the nested `lambda` structure makes a function that makes a function when called. In both cases, the nested `lambda`’s code has access to the variable `x` in the enclosing `lambda`. This works, but it seems fairly convoluted code; in the interest of readability, nested `lambdas` may be best avoided. The good news, perhaps, is that `def` cannot be nested in `lambda`: because `lambda`’s body is an expression, statements like `def` won’t work—thankfully!

Functional Programming Tools

Last up in this chapter’s gumbo is a reprisal of a category of tools we met in earlier in this book, with a few new tips to round out the topic. By most definitions, today’s Python blends support for multiple programming paradigms: *procedural* (with its basic statements), *object-oriented* (with its classes), and *functional*.

The criteria for the latter of these categories are somewhat loose, but by most measures Python’s functional programming toolbox includes its *first-class object* model, nested scope *closures*, and anonymous function *lambdas* that we met earlier; its *generators* and *comprehensions*, which we’ll be expanding on in the next chapter; and perhaps its function and class *decorators* previewed here but fleshed out in this book’s final part.

This toolbox also includes built-in functions that apply other functions to iterables automatically—including functions that call other functions on an iterable’s items (`map`); select items based on a test function (`filter`); and apply functions to pairs of items and running results (`reduce`). Let’s wrap up this chapter with a quick survey of this trio.

Mapping Functions over Iterables: `map`

One of the more common things programs do with lists and other sequences is apply an operation to each item and collect the results—selecting columns in database tables, incrementing pay fields of employees in a company, parsing email attachments, and so on. Python has multiple tools that make such collection-wide operations easy to code. For instance, we’ve seen that updating all the counters in a list can be done easily with a `for` loop:

```
>>> counters = [1, 2, 3, 4]

>>> updated = []
>>> for x in counters:
    updated.append(x + 10)           # Add 10 to each item

>>> updated, counters
([11, 12, 13, 14], [1, 2, 3, 4])
```

But because this is such a common operation, Python also provides built-ins that do most of the work for you. Among them, the `map` function applies a passed-in function to each item in an iterable object and returns a list containing all the function-call results. For example, assuming `counters` is intact:

```
>>> def inc(x): return x + 10      # Function to be run

>>> list(map(inc, counters))       # Collect call results
[11, 12, 13, 14]
```

We met `map` briefly in Chapters 13 and 14, as a way to apply a built-in function to items in an iterable. Here, we make more general use of it by passing in a *user-defined* function to be applied to each item in the list—`map` calls our `inc` on each list item and collects all the return values into a new list. Remember that `map` is a nonsequence iterable, so a `list` call is used to force it to produce all its results for display here (per Chapter 14 coverage).

Because `map` expects a function to be passed in and applied, it also happens to be one of the places where `lambda` commonly appears:

```
>>> list(map((lambda x: x + 3), counters))  # Function expression
[4, 5, 6, 7]
```

Here, the function adds 3 to each item in the `counters` list; as this little function isn’t needed elsewhere, it was written inline as a `lambda`. Because such uses of `map` are equivalent to `for` loops, with a little extra code you can always code a general mapping utility yourself:

```
>>> def mymap(func, iter):
    res = []
    for x in iter: res.append(func(x))
    return res
```

Assuming the function `inc` is still as it was when it was shown previously, we can `map` it across a sequence (or other iterable) with either the built-in or our equivalent:

```
>>> list(map(inc, [1, 2, 3]))           # Built-in is an iterable
[11, 12, 13]
>>> mymap(inc, [1, 2, 3])              # Ours builds a list (see generators)
[11, 12, 13]
```

However, as `map` is a built-in, it's always available, always works the same way, and may have performance benefits (as we'll prove in [Chapter 21](#), it's faster than a manually coded `for` loop in some usage modes). Moreover, `map` can be used in more advanced ways than shown here. For instance, given multiple sequence arguments, it sends items taken from sequences in parallel as distinct arguments to the function:

```
>>> pow(3, 4)                          # 3**4
81
>>> list(map(pow, [1, 2, 3], [2, 3, 4])) # 1**2, 2**3, 3**4
[1, 8, 81]
```

With multiple sequences, `map` expects an N-argument function for N sequences. Here, the `pow` function takes two arguments on each call—one from each sequence passed to `map`. It's not much extra work to simulate this multiple-sequence generality in code, too, but we'll postpone doing so until later in the next chapter, after we've explored some additional iteration tools (hint: it would be better to *generate* items on demand, like the built-in).

The `map` call is also similar to the list comprehension expressions we studied in [Chapter 14](#) and will revisit in the next chapter from a functional perspective:

```
>>> list(map(inc, [1, 2, 3, 4]))
[11, 12, 13, 14]
>>> [inc(x) for x in [1, 2, 3, 4]]
[11, 12, 13, 14]
```

In some cases, `map` may be faster to run than a list comprehension, and it may also require less code. On the other hand, because `map` applies a *function* call to each item instead of an arbitrary *expression*, it is a somewhat less general tool, and often requires extra helper functions or lambdas. Moreover, wrapping a comprehension in parentheses instead of square brackets creates an object that *generates* values on request to save memory and increase responsiveness, much like `map`—a topic we'll take up in the next chapter.

Selecting Items in Iterables: `filter`

The `map` function is a primary and relatively straightforward representative of Python's functional programming toolset. Its close relatives, `filter` and `reduce`, select an iterable's items based on a test function and apply functions to item pairs, respectively.

Because it also returns an iterable, `filter` (like `range`) requires a `list` call to display all its results in a REPL. For example, the following `filter` call picks out items in a sequence that are greater than zero:

```
>>> list(range(-5, 5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(filter((lambda x: x > 0), range(-5, 5)))
[1, 2, 3, 4]
```

We met `filter` briefly earlier in the sidebar “[Why You Will Care: Booleans](#)” on page 308 and while exploring iterables in [Chapter 14](#). Items in the sequence or iterable for which the function returns a true result are added to the result list. Like `map`, this function is roughly equivalent to a `for` loop, but it is built-in, concise, and often fast:

```
>>> res = []
>>> for x in range(-5, 5):
    if x > 0:
        res.append(x)
# The statement equivalent of filter
# Simple but slower today, probably

>>> res
[1, 2, 3, 4]
```

Also like `map`, `filter` can be emulated by *list comprehension* syntax with often simpler results (especially when it can avoid creating a new function), and with a similar *generator expression* when coded with enclosing parentheses to delay production of results—though again, the generator story will have to remain a teaser for the next chapter:

```
>>> [x for x in range(-5, 5) if x > 0]
[1, 2, 3, 4]
```

Combining Items in Iterables: reduce

The functional `reduce` call—once a built-in but now a resident of the `functools` standard-library module—is more complex. It accepts an iterable to process, but it's not an iterable itself: it returns a *single* result that aggregates an iterable's items. To demo, here are two `reduce` calls that compute the sum and product of the items in a list:

```
>>> from functools import reduce
>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

At each step, `reduce` passes the *current* sum or product, along with the *next* item from the list, to the passed-in `lambda` function. By default, the first item in the sequence initializes the starting value. To make that more concrete again, here's the `for` loop equivalent to the first of these calls, with the addition hardcoded inside the loop:

```
>>> L = [1, 2, 3, 4]
>>> res = L[0]
>>> for x in L[1:]:
    res += x

>>> res
10
```

In fact, coding your own reusable and customizable version of `reduce` is fairly straightforward too. The following function emulates most of the built-in's behavior and helps demystify its operation in general:

```
>>> def myreduce(function, sequence):
    tally = sequence[0]
    for next in sequence[1:]:
        tally = function(tally, next)
    return tally

>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> myreduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

The built-in `reduce` also allows an optional third argument, effectively placed before the items in the sequence to serve as an initial value and a default result when the sequence is empty, but we'll leave this extension as a suggested exercise (again, emulating built-in tools is instructive, but superfluous).

If this coding technique has sparked your interest, you might also be interested in the standard-library `operator` module, which provides functions that correspond to built-in expressions and so comes in handy for some uses of functional tools (consult `help` in a REPL or Python's library manual for more details on this module):

```
>>> import operator, functools
>>> functools.reduce(operator.add, [2, 4, 6])           # Function-based +
12
>>> functools.reduce((lambda x, y: x + y), [2, 4, 6])
12
```

Together, `map`, `filter`, and `reduce` support powerful functional programming techniques. As mentioned, many observers would also extend the functional programming toolset in Python to include the nested function closures and anonymous function `lambdas` we've explored, as well as the *generators* and *comprehensions* we've met in piecemeal fashion along the way. To fully grok the latter two, though, we must move on to the next chapter.

Chapter Summary

This chapter's collage took us on a tour of function-related topics: function design guidelines; recursive functions; function attributes, annotations, and decorators; `lambda` expressions; and the `map`, `filter`, and `reduce` built-ins. Some of these are advanced, but most are common in Python programming. The next chapter continues the advanced-topics motif with a reprisal of comprehensions and an unmasking of generators—tools that are just as related to functional programming as to looping statements. Before you move on, though, make sure you've mastered the concepts covered here by working through this chapter's quiz.

Test Your Knowledge: Quiz

1. How are `lambda` expressions and `def` statements related?
2. What's the point of using `lambda`?
3. Compare and contrast `map`, `filter`, and `reduce`.
4. What are function annotations, and how are they used?
5. What are recursive functions, and how are they used?
6. What are some general design guidelines for coding functions?
7. Name three or more ways that functions can communicate results to a caller.

Test Your Knowledge: Answers

1. Both `lambda` and `def` create function objects to be called later. Because `lambda` is an expression, though, it returns a function object instead of assigning it to a name, and it can be used to nest a function definition in places where a `def` will not work syntactically. A `lambda` allows for only a single implicit return value expression, though; because it does not support a block of statements, it is not ideal for larger functions.

2. `lambda` allows us to “inline” small units of executable code, defer its execution, and provide it with state in the form of default arguments and enclosing scope variables. Using a `lambda` is never required; you can always code a `def` instead and reference the function by name. `lambda` comes in handy, though, to embed small pieces of deferred code that are unlikely to be used elsewhere in a program. It commonly appears in callback-based programs such as GUIs, and has a natural affinity with functional tools like `map` and `filter` that expect a processing function.
3. These three built-in functions all apply another function to items in a sequence (or other iterable) object and collect results. `map` passes each item to the function and collects call results, `filter` collects items for which the function returns a true value, and `reduce` computes a single value by applying the function to an accumulator and successive items. Unlike the other two, `reduce` is available in the `functools` module, not the built-in scope (in modern Python history, at least).
4. Function annotations are syntactic embellishments of a function’s arguments and result, which are collected into a dictionary assigned to the function’s `__annotations__` attribute. Python places no semantic meaning on these annotations, but simply packages them for potential use by other tools.
5. Recursive functions call themselves either directly or indirectly in order to loop (i.e., repeat). They may be used to traverse arbitrarily shaped structures, but they can also be used for iteration in general (though the latter role is often more simply and efficiently coded with looping statements). Recursion can often be simulated or replaced by code that uses explicit stacks or queues to have more control over traversals.
6. Functions should generally be small and as self-contained as possible, have a single unified purpose, and communicate with other components through input arguments and return values. They may use mutable arguments to communicate results too if changes are expected, and some types of programs imply other communication mechanisms.
7. Functions can send back results with `return` statements, by changing passed-in mutable arguments, and by setting global variables. Globals are generally frowned upon (except for very special cases, like multithreaded programs) because they can make code more difficult to understand and use. `return` statements are usually best, but changing mutables is fine (and even useful), if expected. Functions may also communicate results with system devices such as files and sockets, but these are beyond our scope here.

Why You Will Care: `lambda` Callbacks

Another common role for `lambda` is to define inline *callback* functions for Python’s `tkinter` GUI API. For example, the following creates a button that prints a message on the console when pressed, assuming `tkinter` is present in your Python (it is by default on most PCs and at least one Android app):

```
from tkinter import Button, mainloop
x = Button(
    text='Press me',
    command=lambda: print('Tapped!'))
x.pack()
mainloop() # This may be optional in some REPLs
```

Here, we register the callback handler by passing a function generated with a `lambda` to the `command` keyword argument. The advantage of `lambda` over `def` in this is that the code that handles a button press is right here, embedded in the button-creation call.

In effect, the `lambda` *defers* execution of the handler until the event occurs: the `print` call happens on button presses, not when the button is created, and effectively “knows” the string it should write when the event occurs. Real GUIs rarely print to consoles, of course, but this demos the coding pattern.

Because the nested function scope rules apply to `lambda`, they automatically see names in the functions in which they are coded and hence don’t require passed-in defaults in most cases (except for loop variables, per [Chapter 17](#)). This is especially useful for accessing the special `self` instance argument that is a local variable in enclosing class method functions (which we’ll study in [Part VI](#), so take this as preview only):

```
class MyGui:
    def makewidgets(self):
        Button(command=lambda: self.onPress('Tapped!'))
    def onPress(self, message):
        ...use message...
```

As you’ll see later, both class objects with `__call__` and *bound methods* often serve in callback roles too—watch for coverage of these in [Chapters 30](#) and [31](#).

And all of this applies to coding event callbacks in other commonly used and portable GUI toolkits for Python—including *Kivy*, *Toga* (in *BeeWare*), *PyQT*, and *wxPython*. `tkinter` gets more press here only because it’s shipped with Python’s standard library. As for all tools, you should vet GUIs for yourself.

Comprehensions and Generations

This chapter explores a set of advanced function-related tools and topics. Its main subjects are *generator functions* and their *generator expression* relatives—user-defined ways to produce results on demand the same way that many built-ins do. Because generators apply the *iteration* protocol and generator expressions reuse *comprehension* syntax, this chapter is also partly a follow-up to [Chapter 14](#) (hence its title). We'll extend these topics to their completion here and demo with larger examples that tie ideas together.

Finally, this chapter provides just enough of an intro to get you started with *async coroutines*—tools that build on generators, but assume knowledge of parallel programming, which is outside the scope of this book and the needs of most Python learners. You won't become an *async* master here, but you'll get a head start for further explorations.

Iteration and generation in Python also encompasses user-defined *classes*, but we'll defer that final part of this story until [Part VI](#), when we study operator overloading. The next chapter continues threads spun here by timing the relative performance of some of this chapter's tools as a larger case study. Before that, though, let's pick up the comprehensions and iterations story where it was last left, and extend it to include value generators.

Comprehensions: The Final Act

As mentioned early in this book, Python supports the procedural, object-oriented, and function programming paradigms. Among these, Python has a host of tools that most observers would consider *functional* in nature, which we enumerated in the preceding chapter—closures, generators, lambdas, comprehensions, maps, decorators, function objects, and more. These tools allow us to apply and combine functions in powerful ways, and often offer state retention and coding solutions that are alternatives to classes and OOP.

For instance, the prior chapter explored tools such as `map` and `filter`—key members of Python's early functional-programming toolset inspired by the Lisp language—that map operations over iterables and collect results. Because this is such a common task in coding, Python eventually sprouted a new expression—the *list comprehension*—a device inspired by languages like Haskell, which is even more flexible than the original functional toolset.

As we’ve seen, list comprehensions apply an arbitrary *expression* to items in an iterable, rather than applying a function. Accordingly, they can be more general tools. In later Pythons, the list comprehension was extended to other roles—sets, dictionaries, and even the value generator expressions we’ll explore in this chapter. Hence, it’s not just for lists anymore, and we’ll simply call it *comprehension* when referring to all its forms collectively.

We first met comprehensions in [Chapter 4](#)’s preview, introduced them in [Part II](#)’s coverage of sets, lists, and dictionaries, and studied them further in [Chapter 14](#), in conjunction with looping statements. Because they’re also related to functional programming tools like the `map` and `filter` calls and repurposed by generator expressions, though, let’s resurrect the topic here for one last look.

List Comprehensions Review

Here’s a brief example to refresh a few neurons. As we learned in [Chapter 7](#), Python’s built-in `ord` function returns the integer code point of a single character. If we wish to collect code points of *all* characters in an entire string, a straightforward approach is to use a simple `for` loop and append the results to a list. In a REPL:

```
>>> res = []
>>> for x in 'text':
    res.append(ord(x))                # Manual results collection

>>> res
[116, 101, 120, 116]
```

Now that we know about `map`, though, we can achieve similar results with a single function call without having to manage list construction in the code:

```
>>> res = list(map(ord, 'text'))      # Apply function to iterable
>>> res
[116, 101, 120, 116]
```

However, we can get the same results from a list comprehension expression—while `map` maps a *function* over an iterable, list comprehensions map an *expression*:

```
>>> res = [ord(x) for x in 'text']   # Apply expression to iterable
>>> res
[116, 101, 120, 116]
```

List comprehensions collect the results of applying an expression to an iterable of values, and return them in a new list. Syntactically, list comprehensions are enclosed in square brackets—to remind you that they construct lists. In their basic form, within the brackets you code an expression that names a variable, followed by what looks like a `for` loop header that names the same variable. You get back the expression’s results for each iteration of the implied loop.

The effect of the preceding example is similar to that of the manual `for` loop and the `map` call. List comprehensions become more convenient, though, when we wish to apply an arbitrary expression instead of a function:

```
>>> [x ** 2 for x in range(10)]       # Squares of numbers 0 to 9
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

To do similar work with a `map` call, we may need to invent a little function to implement the square operation. Because we won’t need this function elsewhere, we’d typically (but not necessarily) code it inline, with a `lambda`:

```
>>> list(map((lambda x: x ** 2), range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This does the same job, and it's only a few keystrokes longer than the equivalent list comprehension. It's also only marginally more complex (at least, once you understand the `lambda`). For more advanced roles, though, list comprehensions will often require considerably less typing.

For instance, as we learned in [Chapter 14](#), you can code an `if` clause after the comprehension's `for` to add selection logic. List comprehensions with `if` clauses can be thought of as analogous to the `filter` built-in in the preceding chapter—they skip an iterable's items for which the `if` clause is not true. As a demo, following are both schemes, along with the equivalent `for`, picking up even numbers from 0 to 9 (`x % 2` is zero for evens only):

```
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]

>>> list(filter((lambda x: x % 2 == 0), range(10)))
[0, 2, 4, 6, 8]

>>> res = []
>>> for x in range(10):
>>>     if x % 2 == 0:
>>>         res.append(x)

>>> res
[0, 2, 4, 6, 8]
```

Though it's penalized by having to code a function to be applied, the `filter` call here is still not much longer than the list comprehension either. However, the list comprehension can *combine* an `if` clause and an arbitrary expression, to give it the effect of a *filter* and a *map*—in a single expression:

```
>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

This time, we collect the squares of the even numbers from 0 through 9: the `for` loop skips numbers for which the attached `if` clause on the right is false, and the expression on the left computes the squares. The equivalent `map` call would require a lot more work on our part—we would have to combine `filter` selections with `map` iteration, making for a noticeably more complex expression:

```
>>> list( map((lambda x: x**2), filter((lambda x: x % 2 == 0), range(10))) )
[0, 4, 16, 36, 64]
```

Formal Comprehension Syntax

In fact, list comprehensions are more general still. In their simplest form, you must always code an accumulation expression and a single `for` clause:

```
[ expression for target in iterable ]
```

Though all other parts are optional, they allow richer iterations to be expressed—you can code any number of nested `for` loops in a list comprehension, and each may have an optional associated `if` test to act as a filter. The general structure of list comprehensions looks like this:

```
[ expression for target1 in iterable1 if condition1
    for target2 in iterable2 if condition2 ...
    for targetN in iterableN if conditionN ]
```

This exact same syntax is inherited by *set* and *dictionary* comprehensions as well as the *generator expressions* coming up, though these use different enclosing characters—curly braces for the first two and often-optional parentheses for the latter—and the dictionary comprehension begins with two expressions separated by a colon (for key and value).

We experimented with the `if` filter clause in the previous section. When `for` clauses are *nested* within a list comprehension, they work like equivalent nested `for` loop statements. For example:

```
>>> res = [x + y for x in [0, 1, 2] for y in [100, 200, 300]]
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

This has the same effect as this substantially more verbose equivalent:

```
>>> res = []
>>> for x in [0, 1, 2]:
    for y in [100, 200, 300]:
        res.append(x + y)

>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Although list comprehensions construct list results, remember that they can iterate over any iterable object. Here's a similar bit of code that traverses strings instead of lists of numbers, and so collects concatenation results:

```
>>> [x + y for x in 'orm' for y in 'ORM']          # 3 * 3 results
['oO', 'oR', 'oM', 'rO', 'rR', 'rM', 'mO', 'mR', 'mM']
```

Each `for` clause can have an associated `if` filter, no matter how deeply the loops are nested—though use cases for the following sort of code, apart perhaps from the multidimensional arrays ahead, start to become more and more difficult to imagine at this level:

```
>>> [x + y for x in 'orm' if x in 'ro' for y in 'ORM' if y > 'M']
['oO', 'oR', 'rO', 'rR']

>>> [x + y + z for x in 'hack' if x > 'c'
    for y in 'CODE' if y in 'OD'
    for z in '123' if z > '1']
['hO2', 'hO3', 'hD2', 'hD3', 'kO2', 'kO3', 'kD2', 'kD3']
```

Finally, here is a similar list comprehension that illustrates the effect of attached `if` selections on nested `for` clauses applied to numeric objects rather than strings:

```
>>> [(x, y) for x in range(5) if x % 2 == 0
    for y in range(5) if y % 2 == 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

This expression combines *even* numbers from 0 through 4 with *odd* numbers from 0 through 4. The `if` clauses filter out items in each iteration. Here is the equivalent statement-based code:

```
>>> res = []
>>> for x in range(5):
    if x % 2 == 0:
        for y in range(5):
            if y % 2 == 1:
                res.append((x, y))

>>> res
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

Recall that if you're confused about what a complex list comprehension does, you can always nest the list comprehension's `for` and `if` clauses inside each other like this—indenting each clause successively further to the right—to derive the equivalent statements. The result is longer, but perhaps clearer in intent to some human readers on first glance, especially those more familiar with basic statements.

The `map` and `filter` equivalent of this last example would be wildly complex and deeply nested, so this book will leave its coding as an exercise for Zen masters, ex-Lisp programmers, and those with far too much free time.

Example: List Comprehensions and Matrixes

Not all list comprehensions are so artificial, of course. Let's look at one more application to stretch our comprehension muscles. As we saw in Chapters 4 and 8, one basic way to code matrixes (a.k.a. multidimensional arrays) in Python is with nested list structures. The following, for example, defines two 3×3 matrixes as lists of nested lists:

```
>>> M = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]

>>> N = [[2, 2, 2],
          [3, 3, 3],
          [4, 4, 4]]
```

Given this structure, we can always index rows, and columns within rows, using normal index operations:

```
>>> M[1]           # Row 2
[4, 5, 6]

>>> M[1][2]        # Row 2, item 3
6
```

List comprehensions are powerful tools for processing such structures because they automatically scan rows and columns. For instance, although this code stores the matrix by rows, to collect the second *column* we can simply iterate across the rows and pull out the desired column, or iterate through positions in the rows and index as we go:

```
>>> [row[1] for row in M]           # Column 2
[2, 5, 8]

>>> [M[row][1] for row in (0, 1, 2)] # Using offsets
[2, 5, 8]
```

Given positions, we can also easily perform tasks such as pulling out a *diagonal*. The first of the following expressions uses `range` to generate the list of offsets and then indexes with the row and column the same, picking out `M[0][0]`, then `M[1][1]`, and so on. The second scales the column index to fetch `M[0][2]`, `M[1][1]`, etc. (we assume the matrix has the same number of rows and columns):

```
>>> [M[i][i] for i in range(len(M))] # Diagonals
[1, 5, 9]

>>> [M[i][len(M)-1-i] for i in range(len(M))]
[3, 5, 7]
```

Changing such a matrix *in place* requires assignment to offsets (use `range` twice if shapes differ):

```
>>> L = [[1, 2, 3], [4, 5, 6]]
>>> for i in range(len(L)):
    for j in range(len(L[i])):
        L[i][j] += 10

>>> L
[[11, 12, 13], [14, 15, 16]]
```

We can't really do the same with list comprehensions, as they make *new lists*, but we could always assign their results to the original name for a similar effect. For example, we can apply an operation to every item in a matrix, producing results in either a simple vector or a matrix of the same shape:

```
>>> [col + 10 for row in M for col in row]           # Assign to M to retain new value
[11, 12, 13, 14, 15, 16, 17, 18, 19]

>>> [[col + 10 for col in row] for row in M]
[[11, 12, 13], [14, 15, 16], [17, 18, 19]]
```

To understand these, translate to their simple statement form equivalents that follow—indent parts that are further to the right in the expression (as in the first loop in the following), and make a new list when comprehensions are nested on the left (like the second loop in the following). As its statement equivalent makes clearer, the second expression in the preceding works because the row iteration is an outer loop: for each row, it runs the nested column iteration to build up one row of the result matrix:

```
>>> res = []
>>> for row in M:                                     # Statement equivalents
    for col in row:                                   # Indent parts further right
        res.append(col + 10)

>>> res
[11, 12, 13, 14, 15, 16, 17, 18, 19]

>>> res = []
>>> for row in M:
    tmp = []                                         # Left-nesting starts new list
    for col in row:
        tmp.append(col + 10)
    res.append(tmp)

>>> res
[[11, 12, 13], [14, 15, 16], [17, 18, 19]]
```

Finally, with a bit of creativity, we can also use list comprehensions to combine values of *multiple matrixes*. The following first builds a flat list that contains the result of multiplying the matrixes pairwise, and then builds a nested list structure having the same values by nesting list comprehensions again:

```
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> N
[[2, 2, 2], [3, 3, 3], [4, 4, 4]]

>>> [M[row][col] * N[row][col] for row in range(3) for col in range(3)]
[2, 4, 6, 12, 15, 18, 28, 32, 36]

>>> [[M[row][col] * N[row][col] for col in range(3)] for row in range(3)]
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

This last expression works because the row iteration is an outer loop again; it's equivalent to this statement-based code:

```
res = []
for row in range(3):
    tmp = []
    for col in range(3):
        tmp.append(M[row][col] * N[row][col])
    res.append(tmp)
```

And for more fun, we can use `zip` to pair items to be multiplied—the following comprehension and loop statement both produce the same list-of-lists pairwise multiplication result as the last preceding example (and because `zip` is a generator of values, this isn't as inefficient as it may seem):

```
[[col1 * col2 for (col1, col2) in zip(row1, row2)] for (row1, row2) in zip(M, N)]

res = []
for (row1, row2) in zip(M, N):
    tmp = []
    for (col1, col2) in zip(row1, row2):
        tmp.append(col1 * col2)
    res.append(tmp)
```

Compared to their statement equivalents, the list comprehension versions here require only one line of code, might run substantially faster for large matrixes, and just might make your head explode. Which brings us to the next section.

When not to use list comprehensions: Code obfuscation

With such generality, list comprehensions can quickly become, well, incomprehensible, especially when nested. Some programming tasks are inherently complex, and we can't sugarcoat them to make them any simpler than they are (see the upcoming permutations for a prime example). Tools like comprehensions are powerful solutions when used wisely, and there's nothing inherently wrong with using them in your scripts.

At the same time, code like that at the end of the prior section may introduce more complexity than it should—and, frankly, tends to disproportionately spark the interest of those holding the darker and misinformed assumption that code obfuscation somehow implies talent. In the interest of software citizenship, some perspective seems in order here.

This book demonstrates advanced comprehensions to teach, but in the real world, programming is not about being *clever and obscure*—it's about how clearly your program communicates its purpose. Writing tricky comprehensions may be a fun academic recreation, but it doesn't work in programs that others will someday need to understand.

In other words, the age-old acronym *KISS* applies here as always: Keep It Simple—traditionally followed either by a word that is now too sexist, or another that is too colorful for a G-rated book like this. If you have to translate code to simpler statements to understand it, it should probably be simpler statements in the first place.

When to use list comprehensions: Speed, conciseness, etc.

Nevertheless, in this case, there is currently a *performance* advantage to the extra complexity: based on tests run under Python today, `map` calls and list comprehensions can be significantly faster than equivalent for loops. This speed difference can vary per usage and Python, but is due to the fact that `map` and list comprehensions run at compiled-language speed inside the interpreter, which is generally faster than running for loop bytecode within the PVM.

Also in the plus column, list comprehensions offer a code *conciseness* that's compelling and even warranted when that reduction in size doesn't also imply a reduction in meaning for the next programmer; many find the *expressiveness* of comprehensions to be a powerful ally; and because `map` and list comprehensions are both expressions, they also can show up syntactically in places that for loop statements cannot, such as in `lambda` and object literals.

Because of all this, list comprehensions and `map` calls are worth knowing and using for simple sorts of iterations, especially if your application's speed is an important consideration. For example, it's hard to argue with the ease and power of code like either of the following:

```
[line.rstrip() for line in open('myfile')]
map((lambda line: line.rstrip()), open('myfile'))
```

To achieve the same memory economy and execution time division as `map`, though, list comprehensions must be coded as *generator expressions*—which is why we've run through this review in the first place, and one of the major topics this chapter turns to next.

Generator Functions and Expressions

Python today supports procrastination much more than it did in the past—it provides tools that produce results only when needed, instead of all at once. We've seen this at work in built-in tools: files that read lines on request, and functions like `map` and `zip` that produce items on demand. Such laziness isn't confined to Python itself, though. In particular, two language constructs delay result creation whenever possible in user-defined operations:

- *Generator functions* are coded as normal `def` statements, but use `yield` statements to return results one at a time, suspending and resuming their state between each.
- *Generator expressions* are similar to the list comprehensions of the prior section, but they return an object that produces results on demand instead of building a result list.

Because neither of these constructs a result list all at once, they both save memory space and allow computation to be split across requests to avoid pauses and enable large (and even “infinite”) results. As you'll see, both of these ultimately perform their delayed-results magic by implementing the *iteration protocol* we studied in [Chapter 14](#).

These features date back to at least Python 2.2 (and were explored even earlier in languages like Icon), and are common in Python code today. Though they may initially seem unusual if you're accustomed to simpler models, you'll probably find generators to be a powerful tool. Moreover, because they are a natural extension to the function, comprehension, and iteration topics we've already explored, you already know more about coding generators than you might expect.

Generator Functions: `yield` Versus `return`

In this part of the book, we've learned about coding normal functions that receive input parameters and send back a single result immediately. It is also possible, however, to write functions that may send back a value and later be resumed, picking up where they left off. Such functions are known as *generator functions* because they generate a series of values over time instead of stopping after just one.

Generator functions are like normal functions in most respects, and in fact are coded with normal `def` statements. However, when created, they are compiled specially into an object that supports the iteration protocol. When called, they don't return a result: they return a result generator that can appear in any iteration context. We studied iterables in [Chapter 14](#), and [Figure 14-1](#) gave a formal and graphic summary of their operation. Here, we'll revisit the subject to see how it applies to generators.

State suspension

Unlike normal functions that return a value and exit, generator functions automatically suspend and resume their execution and state around the point of value generation. Because of that, they are often a

useful alternative to both computing an entire series of values up front and manually saving and restoring state in classes. The *state* that generator functions retain when they are suspended includes both their code location and their entire local scope. Hence, their *local variables* retain information between results, and make it available when the functions are resumed.

The chief code difference between generator and normal functions is that a generator *yields* a value, rather than *returning* one—the `yield` statement suspends the function and sends a value back to the caller, but retains enough state to enable the function to resume from where it left off. When resumed, the function continues execution immediately after the last `yield` run. From the function’s perspective, this allows its code to produce a series of values over time, rather than computing them all at once and sending them back in something like a list.

Iteration protocol integration

To truly understand generator functions, you need to know that they are closely bound up with the notion of the iteration protocol in Python. As we’ve seen, iterator objects define a `__next__` method, which either returns the next item in the iteration, or raises the `StopIteration` exception to end the iteration. An iterable object’s iterator is fetched initially with the `__iter__` method, though this step is a no-op for objects that are their own iterator.

Python for loops, and all other iteration tools, use this iteration protocol to step through a sequence or value generator, if the protocol is supported (if not, iteration falls back on repeatedly indexing sequences instead). Any object that supports this interface works in all iteration tools, and the `iter` and `next` built-ins simplify manual iteration by calling the corresponding double-underscore method (or its internal equivalent).

To support this protocol, functions containing a `yield` statement are compiled specially as *generators*—they are not normal functions, but rather are built to return an object with the expected iteration-protocol methods. When later called, such functions return an iterable object that supports the iteration protocol with an automatically created method named `__next__` to start or resume execution.

Besides `yield`, generator functions may also use a `return` statement that, along with falling off the end of the `def` block, terminates the generation of values by automatically raising a `StopIteration` exception. A generator’s `return` can also give an object that becomes the `value` attribute of the `StopIteration` exception, but it’s ignored by iteration tools and uncommon. From the caller’s perspective, the generator’s `__next__` method simply starts or resumes the function and runs until either the next `yield` result is returned, or a `StopIteration` is raised.

The net effect is that generator functions, coded as `def` statements containing `yield` statements, are automatically made to support the iteration methods protocol and thus may be used in any iteration tool to produce results over time and on demand. The presence of a `yield` in a `def` suffices to make all this happen, and none of this applies to `lambda` because it doesn’t allow statements like `yield` (which is yet another reason to prefer `def`).

Generator functions in action

As usual, this is probably simpler in code than narrative. The following defines a generator function that can be used to produce the squares of a series of numbers—over time:

```
>>> def gensquares(n):
    for i in range(n):
        yield i ** 2          # Resume here later
```

This function yields a value, and so returns to its caller, each time through the loop. When it is resumed, its prior state is restored, including the last values of its variables `i` and `n`, and control picks up again immediately after the `yield` statement. For example, when it's used in the body of a `for` loop, the first iteration starts the function and gets its first result; thereafter, control returns to the function after its `yield` statement each time through the loop:

```
>>> for i in gensquares(5):    # Resume the function
      print(i, end=' ')        # Print last yielded value
```

```
0 1 4 9 16
```

To end the generation of values, functions either use a `return` statement or simply allow control to fall off the end of the function body. As noted, `return` can give a value attached to the exit exception, but usually does not.

To most people, this process seems a bit implicit (if not enchanted) on first encounter. It's actually quite tangible, though. If you really want to see what is going on inside the `for`, call the generator function directly:

```
>>> x = gensquares(3)          # Generator of 0..2 squares
>>> x
<generator object gensquares at 0x10dc6d700>
```

You get back a *generator object* that supports the iteration protocol—the generator function was compiled to return this automatically. The returned generator object in turn has a `__next__` method that starts the function or resumes it from where it last yielded a value, and raises a `StopIteration` exception when the end of the series of values is reached and the function returns. As noted, `next(X)` here calls `X.__next__()` for convenience:

```
>>> next(x)                    # Run to first yield
0
>>> next(x)                    # Resume after yield, run to next yield
1
>>> next(x)                    # Resume after yield, run to next yield
4
>>> next(x)                    # Resume after yield, exception on return
StopIteration
```

Per [Chapter 14](#), `for` loops and other iteration tools work with generators in the same way—by calling the `__next__` method repeatedly, until the exit exception is caught. For generators, the result produces yielded values over time.

Notice that the top-level `iter` call of the iteration protocol isn't required here because generators are their own iterator, supporting just one active iteration scan. To put that another way, generators return themselves for `iter` and support `next` directly. This also holds true in the generator expressions you'll meet later in this chapter:

```
>>> y = gensquares(5)          # Returns a generator which is its own iterator
>>> iter(y) is y                # iter() is not required: a no-op here
True
>>> next(y)                    # Can run next() immediately
0
```

Why generator functions?

Given the simple example we’re using to illustrate fundamentals, you might be wondering just why you’d ever care to code a generator at all. In code so far, for instance, we could simply build the list of result values all at once:

```
>>> def buildsquares(n):
    res = []
    for i in range(n): res.append(i ** 2)
    return res

>>> for x in buildsquares(5): print(x, end=' ')

0 1 4 9 16
```

For that matter, we could use any of the for loop, map, or list comprehension techniques we’ve already mastered:

```
>>> for x in [n ** 2 for n in range(5)]:
    print(x, end=' ')

0 1 4 9 16

>>> for x in map((lambda n: n ** 2), range(5)):
    print(x, end=' ')

0 1 4 9 16
```

However, generators can be better in terms of both memory use and performance in larger programs. They allow functions to avoid doing all the work up front, which is especially useful when the result lists are large or when it takes a lot of computation to produce each value. Generators distribute the time required to produce the series of values among loop iterations, and can even produce a series of values that has no end at all (an “infinite” result).

Moreover, for more advanced uses, generators can provide a simpler alternative to manually saving the state between iterations in class objects—with generators, variables accessible in the function’s scopes are saved and restored automatically. You’ll be able to judge this contrast after we discuss class-based iterables in more detail in [Part VI](#).

Generator functions are also more broadly focused than implied so far. They can operate on and return any type of object, and as *iterables* may appear in any of [Chapter 14](#)’s iteration tools, including tuple calls, enumerations, and dictionary comprehensions:

```
>>> def ups(line):
    for sub in line.split(','):          # Substring generator
        yield sub.upper()

>>> tuple(ups('aaa,bbb,ccc'))           # All iteration contexts
('AAA', 'BBB', 'CCC')

>>> {i: s for (i, s) in enumerate(ups('aaa,bbb,ccc'))}
{0: 'AAA', 1: 'BBB', 2: 'CCC'}
```

In a moment you’ll observe the same assets for generator expressions—a tool that trades function flexibility for comprehension conciseness. Later in this chapter you’ll also see that generators can sometimes enable otherwise impractical tasks, by producing components of result sets that would be far too large to create all at once. First, though, let’s explore some advanced generator function features.

Extended generator function protocol: send versus next

Somewhere along generators' evolutionary path (in Python 2.5), a `send` method was added to the generator function protocol. The `send` method advances to the next item in the series of results, just like `__next__`, but also provides a way for the caller to communicate with the generator, and hence to affect its operation.

Technically, `yield` is an *expression* form that returns the item passed to `send`, not a statement. It can be coded either way (and usually is a statement), but when used as an expression must be enclosed in parentheses unless it's the only item on the right side of an assignment statement. For example, `X = yield Y` is OK, as is `X = (yield Y) + Z`.

When this extra protocol is used, values are sent into a generator `G` by calling `G.send(value)`. The generator's code is then resumed, and the `yield` expression inside the generator returns the value passed to `send` by the caller. If the regular `G.__next__()` method (or its `next(G)` equivalent) is called to advance, the `yield` simply returns `None`. For example:

```
>>> def gen():
    for i in range(10):
        x = yield i
        print('=>', x)

>>> G = gen()
>>> next(G)           # Must call next() first, to start generator
0
>>> G.send(77)        # Advance, and send value to yield expression
=> 77
1
>>> G.send(88)
=> 88
2
>>> next(G)           # next() and X.__next__() send None
=> None
3
```

The `send` method can be used, for example, to code a generator that its caller can terminate by sending a termination code, or redirect by passing a new position in data being processed inside the generator.

In addition, generators also support a `throw(type)` method to raise an exception inside the generator at the latest `yield`, and a `close` method that raises a special `GeneratorExit` exception inside the generator to terminate the iteration entirely. Together with `send`, these are advanced features added to make generators more like a tool called coroutines, a role eventually subsumed in part by the upcoming `async`. Hence, we won't delve further here; see Python's standard manuals for more information, and watch for more on exceptions in [Part VII](#).

Note, though, that while Python provides a `next(X)` convenience built-in that calls the `X.__next__()` method of an object, other generator methods, like `send`, must be called as methods of generator objects directly (e.g., `G.send(X)`). This makes sense if you realize that these extra methods are implemented on built-in generator objects only, whereas the `__next__` method applies to all iterable objects—both built-in types and user-defined classes.

The yield from extension

Even later, Python 3.3 introduced extended syntax for the `yield` statement with a *from generator* clause that allows generators to delegate to nested generators (known as *subgenerators*). In simple cases,

it's the equivalent to a yielding for loop. As a demo, the `list` call in the following forces a generator to produce values from two sources:

```
>>> def both(N):
    for i in range(N): yield i
    for i in map(lambda x: x ** 2, range(N)): yield i

>>> list(both(5))
[0, 1, 2, 3, 4, 0, 1, 4, 9, 16]
```

The `yield from` syntax makes this more concise and explicit, and supports all the usual generator usage contexts. In the following, `both` is called a *delegating* generator, and the `range` and `map` built-ins serve as subgenerators (`join` also uses a generator expression here: see the next section):

```
>>> def both(N):
    yield from range(N)
    yield from map(lambda x: x ** 2, range(N))

>>> list(both(5))
[0, 1, 2, 3, 4, 0, 1, 4, 9, 16]

>>> ' : '.join(str(i) for i in both(5))
'0 : 1 : 2 : 3 : 4 : 0 : 1 : 4 : 9 : 16'
```

The `yield from` also supports arbitrary *chains* of generators. In the following, for example, two generators `yield from` others, the last of which ultimately uses `yield` to send results up through the chain:

```
>>> def c(n):
    yield n * 8                # The bottom of the "chain"
    yield n ** 2

>>> def b(n):
    yield from c(n)           # Delegate to subgenerator

>>> def a(n):
    yield from b(n)           # Delegate to subgenerator

>>> g = a(4)                  # Make top generator, fetch results
>>> g
<generator object a at 0x10bd17940>
>>> next(g)
32

>>> for i in a(4): print(i)    # Force results production
32
16
```

In more advanced roles, though, this extension allows subgenerators to receive *sent* and *thrown* values directly from the delegating generator's caller, and return a final value to the delegating generator as the result of the `yield from` expression. The net effect allows generators to be split into multiple subgenerators much as a single function can be split into multiple subfunctions, but still operate as though their code appeared inline where the `yield from` appears.

Since this is both uncommon and beyond this chapter's scope, we'll again defer to Python's standard manuals for more details. For an additional `yield from` example, see the solution to Exercise 11 from ["Test Your Knowledge: Part IV Exercises" on page 554](#), and stay tuned for a glimpse of `async def` coroutines ahead—whose `await` is something like a generator `yield from`, with a delegation to an event loop to allow other tasks to be run during pauses. Here, let's move on to a tool close enough to `yield` to be called a fraternal twin.

Generator Expressions: Iterables Meet Comprehensions

Because the delayed evaluation of generator functions was so useful, later Pythons eventually combined the notions of iterables and comprehensions in a new tool: *generator expressions*. Syntactically, generator expressions are just like normal list comprehensions, and support all their syntax—including if filters and for loop nesting—but they are enclosed in parentheses instead of square brackets (like tuples, their enclosing parentheses are often optional):

```
>>> [x ** 2 for x in range(5)]          # List comprehension: build a list
[0, 1, 4, 9, 16]

>>> (x ** 2 for x in range(5))          # Generator expression: make an iterable
<generator object <genexpr> at 0x109607ac0>
```

In fact, at least on a functionality basis, coding a list comprehension is essentially the same as wrapping a generator expression in a list built-in call to force it to produce all its results in a list at once:

```
>>> list(x ** 2 for x in range(5))      # List comprehension equivalence
[0, 1, 4, 9, 16]
```

Operationally, however, generator expressions are very different: instead of building the result list in memory, they return a *generator object*—an automatically created iterable. This iterable object in turn supports the *iteration protocol* to yield one piece of the result list at a time in any iteration tool. The iterable object also retains generator state while active—the variable *x* in the preceding expressions, along with the generator’s code location.

The net effect is much like that of generator functions, but in the context of a comprehension *expression*: we get back an object that remembers where it left off after each part of its result is returned. Also like generator functions, looking under the hood at the protocol that these objects automatically support can help demystify them; the *iter* call is again not required at the top here, for reasons we’ll expand on ahead:

```
>>> G = (x ** 2 for x in range(3))
>>> iter(G) is G                        # iter(G) optional: __iter__ returns self
True
>>> next(G)                             # Generator objects: automatic methods
0
>>> next(G)
1
>>> next(G)
4
>>> next(G)
StopIteration

>>> G
<generator object <genexpr> at 0x109607920>
```

Again, we don’t typically see the next iterator machinery under the hood of a generator expression like this because for loops and similar tools trigger it for us automatically:

```
>>> for num in (x ** 2 for x in range(3)):    # Calls next() automatically
    print(num, num / 2.0, sep=', ')

0, 0.0
1, 0.5
4, 2.0
```

As we’ve already seen, every iteration tool does this—including for loops; the `list` call we used earlier; comprehensions of all kinds; the `map` and `filter` functional tools; and other iteration tools we explored in [Chapter 14](#), including the `sum`, `sorted`, `any`, and `all` built-in functions. As *iterables*, generator expressions can be used in any of these iteration tools, just like both physical sequences and the results of a generator-function call.

For example, the following deploys generator expressions in the string `join` method call and tuple assignment, iteration tools both. Recall that `join` with an empty-string delimiter concatenates values produced by its iterable:

```
>>> ''.join(x.upper() for x in 'aaa,bbb,ccc'.split(','))
'AAABBBCCC'

>>> a, b, c = (x + '\n' for x in 'aaa,bbb,ccc'.split(','))
>>> a, c
('aaa\n', 'ccc\n')
```

Notice how the `join` call in the preceding doesn’t require *extra* parentheses around the generator. Syntactically, parentheses are *not required* around a generator expression that is the sole item already enclosed in parentheses used for other purposes—like those of a function call. Parentheses are required in all other cases, however, even if they seem extra, as in the second call to `sorted` that follows:

```
>>> sum(x ** 2 for x in range(3))                                # Parens optional
5
>>> sorted(x ** 2 for x in range(3))                             # Parens optional
[0, 1, 4]
>>> sorted((x ** 2 for x in range(3)), reverse=True)            # Parens required!
[4, 1, 0]
```

Like the often-optional parentheses in tuples, there is no widely accepted rule on this, though a generator expression does not have as clear a role as a fixed collection of other objects as a tuple, making extra parentheses seem perhaps more spurious here.

Why generator expressions?

Just like generator functions, generator expressions are a *memory-space* optimization—they do not require the entire result list to be constructed all at once, as the square-bracketed list comprehension does. Also like generator functions, they divide the work of results production into smaller *time slices*—they yield results in piecemeal fashion, instead of making the caller wait for the full set to be created in a single call.

On the other hand, because generator expressions often run *slower* today than list comprehensions, they may be best reserved for result sets that are very large, or programs that cannot wait for full results generation. A more authoritative statement about performance, though, will have to await the timing scripts we’ll code in the next chapter.

On the subjective upside, generator expressions also offer significant *coding* advantages—as the next sections show.

Generator expressions versus map

One way to see the coding benefits of generator expressions is to compare them to other functional tools, as we did for list comprehensions. For example, generator expressions often are equivalent to `map` calls, because both generate result items on request. Like list comprehensions, though, generator expressions may be simpler to code when the operation applied is not a function call:

```
>>> list(map(abs, (-1, -2, 3, 4)))           # Map function on tuple
[1, 2, 3, 4]
>>> list(abs(x) for x in (-1, -2, 3, 4))     # Generator expression
[1, 2, 3, 4]

>>> list(map(lambda x: x * 2, (1, 2, 3, 4)))  # Nonfunction case
[2, 4, 6, 8]
>>> list(x * 2 for x in (1, 2, 3, 4))        # Simpler as generator?
[2, 4, 6, 8]
```

The same holds true for text-processing use cases like the `join` call we saw earlier—a list comprehension makes an extra temporary list of results, which is completely *pointless* in this context because the list is not retained, and `map` loses simplicity points compared to generator expression syntax when the operation being applied is not a call:

```
>>> line = 'aaa,bbb,ccc'
>>> ''.join([x.upper() for x in line.split(',')])  # Makes temporary list
'AAABBBCCC'

>>> ''.join(x.upper() for x in line.split(','))    # Generates results
'AAABBBCCC'
>>> ''.join(map(str.upper, line.split(',')))        # Generates results
'AAABBBCCC'

>>> ''.join(x * 2 for x in line.split(','))        # Simpler as generator?
'aaaaaabbbbbcccccc'
>>> ''.join(map(lambda x: x * 2, line.split(',')))
'aaaaaabbbbbcccccc'
```

Both `map` and generator expressions can also be arbitrarily *nested*; when coded this way, a `list` call or other iteration tool starts the entire process of producing results. For example, the list comprehension in the following produces the same result as the `map` and generator equivalents that follow it, but makes two physical lists. The others generate just one integer at a time with nested generators, and the generator expression form may more clearly reflect its intent:

```
>>> [x * 2 for x in [abs(x) for x in (-1, -2, 3, 4)]]  # Nested list comps
[2, 4, 6, 8]

>>> list(map(lambda x: x * 2, map(abs, (-1, -2, 3, 4)))) # Nested maps
[2, 4, 6, 8]

>>> list(x * 2 for x in (abs(x) for x in (-1, -2, 3, 4))) # Nested generators
[2, 4, 6, 8]
```

Although the effect of all three of these is to combine operations, the generators do so without making multiple temporary lists. In fact, the next example both nests *and* combines generators—the nested generator expression is activated by `map`, which in turn is only activated by `list`:

```
>>> import math
>>> list(map(math.sqrt, (x ** 2 for x in range(4))))    # Nested combos
[0.0, 1.0, 2.0, 3.0]
```

Technically speaking, the `range` on the right in the preceding is a value generator too, activated by the generator expression itself—forming *three levels* of value generation, which produce individual values from inner to outer only on request, and which “just works” because of Python’s iteration tools and protocol. In fact, generator nestings can be arbitrarily mixed and deep, though some may be more valid than others:

```
>>> list(map(abs, map(abs, map(abs, (-1, 0, 1)))))           # Nesting gone bad?
[1, 0, 1]
>>> list(abs(x) for x in (abs(x) for x in (abs(x) for x in (-1, 0, 1))))
[1, 0, 1]
```

These last examples illustrate how general generators can be, but are also coded in an intentionally complex form to underscore that generator expressions have the same potential for abuse as the list comprehensions discussed earlier—as usual, you should keep them simple unless they must be complex.

When used well, though, generator expressions combine the expressiveness of list comprehensions with the space and time benefits of other iterables. The following *nonnested* alternatives, for example, provide simpler solutions to the preceding three listings’ nested codings, but still leverage generators’ strengths:

```
>>> list(abs(x) * 2 for x in (-1, -2, 3, 4))                 # Same, unnested
[2, 4, 6, 8]
>>> list(math.sqrt(x ** 2) for x in range(4))                # Flat may be better
[0.0, 1.0, 2.0, 3.0]
>>> list(abs(x) for x in (-1, 0, 1))
[1, 0, 1]
```

Generator expressions versus filter

Generator expressions also support all the usual list comprehension syntax—including *if* clauses, which work like the *filter* call we met earlier. Because *filter* is an iterable that generates its results on request, a generator expression with an *if* clause is operationally equivalent. Again, the *join* in the following is an iteration tool that suffices to force all forms to produce their results:

```
>>> line = 'aa bbb c'
>>> ''.join(x for x in line.split() if len(x) > 1)           # Generator with if
'aabbb'
>>> ''.join(filter(lambda x: len(x) > 1, line.split()))        # Equivalent filter
'aabbb'
```

The generator seems just marginally simpler than the *filter* here. As for list comprehensions, though, adding processing steps to *filter* results requires a *map* too, which makes *filter* noticeably more complex:

```
>>> ''.join(x.upper() for x in line.split() if len(x) > 1)
'AABBB'
>>> ''.join(map(str.upper, filter(lambda x: len(x) > 1, line.split())))
'AABBB'
```

In effect, generator expressions provide more general coding structures that do not rely on functions, but still delay results production. Also like list comprehensions, there is always a statement-based equivalent to a generator expression, though it sometimes renders substantially more code:

```
>>> res = ''
>>> for x in line.split():                                     # Statement equivalent?
    if len(x) > 1:                                           # This is also a join
        res += x.upper()

>>> res
'AABBB'
```

In this case, though, the statement form isn't quite the same—it cannot produce items one at a time, and it's also emulating the effect of the `join` that forces results to be produced all at once. The true equivalent to a generator expression would be a generator function with a `yield`, as the next section will show.

Generator Functions Versus Generator Expressions

Let's recap what we've covered so far in this section:

Generator functions

A function `def` statement that contains a `yield` statement is turned into a generator function. When called, it returns a new *generator object* with automatic retention of local scope and code position; an automatically created `__iter__` method that simply returns itself; and an automatically created `__next__` method that starts the function or resumes it where it last left off, and raises `StopIteration` when finished producing results.

Generator expressions

A comprehension expression enclosed in parentheses is known as a generator expression. When run, it returns a new *generator object* with the same automatically created method interface and state retention as a generator function call's results—with an `__iter__` method that simply returns itself; and a `__next__` method that starts the implied loop or resumes it where it last left off, and raises `StopIteration` when finished producing results.

The net effect is to automatically produce results on demand in all iteration tools that employ either of these.

As implied by preceding coverage, the same iteration can often be coded with *either* a generator function or a generator expression. The following generator *expression*, for example, repeats each character in a string four times:

```
>>> G = (c * 4 for c in 'hack')           # Generator expression
>>> list(G)                               # Force generator to produce results
['hhhh', 'aaaa', 'cccc', 'kkkk']
```

The equivalent generator *function* requires slightly more code, but its multiple-statement block will be able to code more logic and use more state information if needed. In fact, this is essentially the same as the prior chapter's trade-off between `lambda` and `def`—expression conciseness versus statement power:

```
>>> def timesfour(S):                     # Generator function
    for c in S:
        yield c * 4

>>> G = timesfour('hack')
>>> list(G)                               # Iterate automatically
['hhhh', 'aaaa', 'cccc', 'kkkk']
```

To their clients, the two are more similar than different. For instance, both expressions and functions support both automatic and manual iteration—the prior `list` call iterates automatically, and the following iterate manually:

```
>>> G = (c * 4 for c in 'hack')
>>> I = iter(G)                            # Iterate manually (expression)
>>> next(I)
'hhhh'
>>> next(I)
'aaaa'
```

```
>>> G = timesfour('hack')
>>> I = iter(G)                                # Iterate manually (function)
>>> next(I)
'hhhh'
>>> next(I)
'aaaa'
```

In either case, Python automatically creates a generator object, which has both the methods required by the iteration protocol, and state retention for variables in the generator's code and its current code location. Notice how we make new generators here to iterate again—as explained in the next section, generators are single-pass iterators.

First, though, here's the true statement-based equivalent of the expression at the end of the prior section: a function that yields values—though the difference is irrelevant if the code using it produces all results with a tool like `join`:

```
>>> line = 'aa bbb c'

>>> ''.join(x.upper() for x in line.split() if len(x) > 1)    # Expression
'AABBB'

>>> def gensub(line):                                          # Function
    for x in line.split():
        if len(x) > 1:
            yield x.upper()

>>> ''.join(gensub(line))                                      # But why generate?
'AABBB'
```

While generators have valid roles, in cases like this the use of generators over the simple statement equivalent shown earlier may be difficult to justify, except on stylistic grounds: if you're just going to immediately `join` generated results anyhow, you might as well skip generators and use simple loops. On the other hand, trading four lines for the generator expression's one may to many seem fairly compelling stylistic grounds!

Generator Odds and Ends

This section wraps up generators with a quick rundown of associated but lesser topics. After this, we'll move on to larger examples, but make sure you have a handle on the smaller bits before jumping ahead.

Generators are single-pass iterables

First up, a subtle but important point: both generator functions and generator expressions are their own iterators and thus support just *one active iteration*—unlike some built-in types, you can't have multiple iterators of either generator positioned at different locations in the stream of results. Because of this, a generator's iterator is the generator itself; in fact, as suggested earlier, calling `iter` on a generator expression or function is an optional no-op:

```
>>> G = (c * 4 for c in 'hack')
>>> iter(G) is G                                             # My iterator is myself: G has __next__
True
```

If you do iterate over the results stream manually with multiple iterators, they will all point to the same position:

```

>>> G = (c * 4 for c in 'hack')           # Make a new generator
>>> I1 = iter(G)                         # Iterate manually
>>> next(I1)
'hhhh'
>>> next(I1)
'aaaa'
>>> I2 = iter(G)                         # Second iterator at same position!
>>> next(I2)
'cccc'

```

Moreover, once any iteration runs to completion, all are exhausted—we have to make a new generator to start again:

```

>>> list(I1)                             # Collect the rest of I1's items
['kkkk']
>>> next(I2)                             # Other iterators exhausted too
StopIteration

>>> I3 = iter(G)                         # Ditto for new iterators
>>> next(I3)
StopIteration

>>> I3 = iter(c * 4 for c in 'hack')     # New generator to start over
>>> next(I3)
'hhhh'

```

The same holds true for generator functions—the following `def` statement-based equivalent supports just one active iterator and is exhausted after one pass:

```

>>> def timesfour(S):
    for c in S:
        yield c * 4

>>> G = timesfour('hack')                # Generator functions work the same way
>>> iter(G) is G                          # One scan per generator (call)
True
>>> I1, I2 = iter(G), iter(G)
>>> next(I1)
'hhhh'
>>> next(I1)
'aaaa'
>>> next(I2)                             # I2 at same position as I1
'cccc'

```

This is different from the behavior of some built-in objects like lists, which support multiple iterators and passes and even reflect their in-place changes in active iterators:

```

>>> L = [1, 2, 3, 4]
>>> I1, I2 = iter(L), iter(L)
>>> next(I1)
1
>>> next(I1)
2
>>> next(I2)                             # Lists support multiple iterators
1
>>> del L[2:]                             # Changes reflected in iterators
>>> next(I1)
StopIteration

```


Though not readily apparent in these simple examples, this can matter in your code: if you wish to scan a generator's values multiple times, you must either create a *new* generator for each scan or build a rescannable *list* out of its values—a single generator's values will be consumed and exhausted after a single pass. See this chapter's sidebar “[Why You Will Care: Iteration Versus Python Morph](#)” on page 526 for a prime example of the sort of code impacted by this.

When we begin coding *class-based* iterables in [Part VI](#), you'll also see that it's up to us to decide how many iterations we wish to support for our objects, if any. In general, objects that wish to support multiple scans will return supplemental class objects instead of themselves for `iter`. The next section previews more of this model.

Generation in built-ins and classes

Although we've focused on coding value generators ourselves in this section, don't forget that many built-in types behave the same way. As we saw in [Chapter 14](#), for example, *dictionaries* and *files* generate results too:

```
for key in dictionary: ...                # See Chapter 14 et al.
for line in file: ...
```

Though beyond this book's scope, many Python standard-library tools generate values too, including its *directory walker*—which at each level of a folder tree yields a tuple of the current directory, its subdirectories, and its files:

```
>>> import os
>>> for (root, subs, files) in os.walk('.'):    # Directory-walk generator
        for name in files:                    # A Python 'find' operation
            if name.endswith('.py'):          # Also os.path.join(root, name)
                print(root, name)

../Chapter02 script0.py
../Chapter03 myfile.py
../Chapter03 script1.py
...etc...
```

Because the current Python-coded implementation of `os.walk` uses `yield` to return results, it's a normal generator function, and hence iterable object, that generates a three-item tuple on each iteration:

```
>>> G = os.walk('.')                        # A single-scan iterator: iter(G) optional
>>> next(G)
('.', ['Chapter02', 'Chapter03', 'Chapter04', '...etc...', 'Chapter37'], [])
>>> next(G)
('../Chapter02', ['__pycache__', 'script0.py'])
```

By yielding results as it goes, the walker does not require its clients to wait for an entire tree to be scanned. See Python's manuals for more on this tool. For system-tools fans, also see the next chapter for an example that uses `os.popen`—a related iterable used to run a shell command and read its output. And for additional examples of built-in value generators and the iteration tools that use them, review [Chapter 14](#).

Finally, although beyond the scope of this chapter, it is also possible to implement arbitrary user-defined generator objects with *classes* that conform to the iteration protocol. Such classes define `__iter__` and `__next__` methods explicitly to support the protocol:

```
class SomeIterable:
    def __iter__(...): ...    # On iter(): return self or supplemental object
    def __next__(...): ...    # On next(): coded here, or in another class
```

As the prior section suggested, these classes usually return their objects directly for single-iteration behavior, or a supplemental object with scan-specific state for multiple-scan support.

Alternatively, a user-defined iterable class's `__iter__` may use `yield` to transform itself into a generator, and a `__getitem__` indexing method is also available as a fallback option for iteration with trade-offs we'll explore later. However this is coded, the iterator and generator story won't really be complete until we've seen how it also maps to classes. For now, we'll have to settle for postponing its conclusion—and its final sequel—until [Chapter 30](#).

Comprehensions versus type calls and generators

We've also been focusing on list comprehensions and generators in this chapter, but keep in mind that there are two other comprehension expression forms: the set and dictionary comprehensions we met earlier in [Chapter 14](#) and [Part II](#). For reference and closure, here's a summary of all the comprehension forms in Python:

```
>>> [x * x for x in range(10)]           # List comprehension: builds list
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]    # Fixed order, made all at once

>>> (x * x for x in range(10))          # Generator expression: yields items
<generator object <genexpr> at 0x100a1f920>

>>> {x * x for x in range(10)}           # Set comprehension: builds set
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}    # Random order, no duplicates

>>> {x: x * x for x in range(10)}        # Dict comprehension: insert order
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

All of these forms use the same formal comprehension syntax we met earlier, but their enclosing delimiters (and key, for dictionaries) denote their differing roles. In a sense, list, set, and dictionary comprehensions are syntactic sugar for passing generator expressions to type names. Because all accept any iterable, a generator works here too:

```
>>> list(x ** 2 for x in range(10))      # Generator plus type name
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> set(x * x for x in range(10))        # Similar to a set comprehension
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}

>>> dict((x, x * x) for x in range(10))
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

But don't read too much into such equivalences. Because implementations may vary arbitrarily, you should generally collect performance data before adopting an alternative. In this case, generators are actually slower than the equivalent list comprehension today, as we'll prove in the next chapter. Programs that run lots of them may need to care.

Scopes and comprehension variables

Now that we've seen all comprehension forms, [Chapter 17](#)'s overview of the localization of loop variables in these expressions may make more sense. In all forms, the loop variable (or variables) after the `for` is local to the expression—it won't clash with names outside, but is also not available there, and this differs from `for` statements:

```

>>> X = 99
>>> [X for X in range(5)]      # All comprehensions localize loop variables
[0, 1, 2, 3, 4]
>>> X                          # Enclosing-scope X was not changed
99
>>> for X in range(5): pass    # But loop statements do not localize!
>>> X
4

```

As noted in [Chapter 17](#), loop variables assigned in a comprehension really live in a further nested special-case scope, but other names referenced within these expressions follow the usual LEGB rules. In the following generator, for example, *Z* is localized in the comprehension, but *Y* and *X* are found in the enclosing local and global scopes as usual:

```

>>> X = 'aaa'
>>> def func():
    Y = 'bbb'
    print('-'.join(Z for Z in X + Y))  # Z=comprehension, Y=local, X=global

>>> func()
a-a-a-b-b-b

```

One exception here: names assigned by the `:=` expression inside a comprehension do leak out of comprehension, and generally behave as though they were assigned in the scope containing the comprehension itself:

```

>>> S = 'hack'
>>> [(temp := S[i]) + temp.upper() for i in range(len(S))]
['hH', 'aA', 'cC', 'kK']
>>> temp
'k'
>>> i
NameError: name 'i' is not defined. Did you mean: 'id'?

```

While this allows comprehensions to both reuse and export values, keep in mind that a `lambda` container’s local scope effectively plugs the leak—`temp` in the following, for example, lives only in the `lambda`’s scope:

```

>>> del temp
>>> f = lambda: [(temp := S[i]) + temp.upper() for i in range(len(S))]
>>> f()
['hH', 'aA', 'cC', 'kK']
>>> temp
NameError: name 'temp' is not defined

```

We’ll code such a `:=` nesting in a sequence-scrambler example ahead, though some observers may deem it much less transparent than simple assignment statements, and other real-world roles must await your discovery. See [Chapter 11](#) for more background on `:=` named assignment.

Generating “infinite” (well, indefinite) results

Finally, it was noted earlier in passing that generators can even produce “infinite” results that tools like list comprehensions cannot. This may sound more impressive than it is; really, this just means that a generator can keep yielding results from its retained state’s local variables *indefinitely*—until its client grows tired of them:

```

>>> def squares(n):
    while True:
        yield n ** 2
        n += 1
    # Generate results "forever"
    # Or until no more next calls...

>>> G = squares(2)
>>> next(G)
4
>>> next(G)
9
>>> for i in range(10): print(next(G), end=' ')

16 25 36 49 64 81 100 121 144 169

```

This pattern may be useful in limited roles like test-parameter generation, and other tools can't compete in this event; list comprehensions, for example, must collect all results at once. In the end, though, this is a narrow role, generators are mortal, and so are we—so let's move on to some code that's a bit more tangible in the next section.

Example: Shuffling Sequences

To demonstrate the power of iteration and generation tools in action, let's turn to some more complete examples. In [Chapter 18](#), we wrote a testing function, based on earlier code in [Chapter 13](#), that scrambled the order of arguments used to verify generalized intersection and union functions. There, it was noted that this might be better coded as a generator of values. Now that we've learned how to write generators, this serves to illustrate a practical application.

One note up front: because they slice and concatenate objects, all the examples in the section (including the permutations at the end) work only on *sequences* like strings and lists, not on arbitrary *iterables* like files, maps, and other generators. That is, some of these examples will *be* generators themselves, producing values on request, but they cannot process generators as their *inputs*. Generalization for broader categories is left as an open issue, though the code here will suffice unchanged if you wrap nonsequence generators in `list` calls before passing them in.

Scrambling Sequences

First, let's review. As coded in [Chapter 18](#), we can reorder a sequence with slicing and concatenation, moving the front item to the end on each loop; *slicing* instead of indexing the item allows `+` to work for arbitrary sequence types:

```

>>> L, S = [1, 2, 3], 'code'

>>> for i in range(len(S)):
    S = S[1:] + S[:1]
    print(S, end=' ')

odec deco ecod code

>>> for i in range(len(L)):
    L = L[1:] + L[:1]
    print(L, end=' ')

[2, 3, 1] [3, 1, 2] [1, 2, 3]

```

Alternatively, as we also saw in [Chapter 13](#), we get the same results by moving an entire front section to the end, though the order of the results varies slightly:

```
>>> for i in range(len(S)):           # Coding 2: for positions 0..3
    X = S[i:] + S[:i]               # Rear part + front part (same effect)
    print(X, end=' ')

code odec deco ecod
```

Trace this code to see how it works; each version produces N results for an N -item sequence passed in.

Simple functions

As is, this code works on specific named variables only, and simply prints its result. As we've seen, it's easy to generalize this by turning it into a normal *function* that can both work on any object passed to its argument and return its result for arbitrary use. The following does so for the second coding alternative; since its first cut exhibits the classic list comprehension pattern, we can also save some work by coding it as such in the second:

```
>>> def scramble(seq):
    res = []
    for i in range(len(seq)):
        res.append(seq[i:] + seq[:i])
    return res

>>> scramble('code')
['code', 'odec', 'deco', 'ecod']

>>> def scramble(seq):
    return [seq[i:] + seq[:i] for i in range(len(seq))]

>>> scramble('code')
['code', 'odec', 'deco', 'ecod']

>>> for x in scramble((1, 2, 3)):
    print(x, end=' ')

(1, 2, 3) (2, 3, 1) (3, 1, 2)
```

We could use recursion here as well, but it's probably overkill in this fixed and linear context.

Generator functions

The preceding section's simple approach works, but must build an entire result list in memory all at once (not great on memory usage if it's massive), and requires the caller to wait until the entire list is complete (less than ideal if this takes a substantial amount of time). You should know by now that we can do better on both fronts by translating this to a *generator function* that yields one result at a time, using either coding scheme:

```
>>> def scramble(seq):
    for i in range(len(seq)):
        seq = seq[1:] + seq[:1]           # Generator function, coding 1
        yield seq                         # Assignments work here

>>> def scramble(seq):
    for i in range(len(seq)):
        yield seq[i:] + seq[:i]          # Generator function, coding 2
                                         # Yield one item per iteration
```

Both of these alternatives produce values when used by an iteration tool like `list` or `for`, though the second version's results order (shown here) again varies slightly in ways that probably won't matter to future clients:

```

>>> list(scramble('code'))                # list() generates all results
['code', 'odec', 'deco', 'ecod']

>>> list(scramble((1, 2, 3)))              # Any sequence type works
[(1, 2, 3), (2, 3, 1), (3, 1, 2)]

>>> for x in scramble((1, 2, 3)):          # for loops generate results
    print(x, end=' ')

(1, 2, 3) (2, 3, 1) (3, 1, 2)

```

Both generator functions retain their local scope state (seq and i) while active, minimize memory space requirements, and divide the work into shorter time slices. As full functions, they are also very general. Moreover, because iteration tools work the same whether stepping through a real list or a generator of values, the function can select between the two codings freely, and even change strategies in the future without impacting callers.

Generator expressions

As we've also seen, *generator expressions*—comprehensions in parentheses instead of square brackets—also generate values on request and retain their local state. As expressions, they're not as flexible as full functions, but because they yield their values automatically, generator expressions can often be more concise in specific use cases like this:

```

>>> S = 'code'
>>> G = (S[i:] + S[:i] for i in range(len(S)))    # Generator expr, coding 2
>>> list(G)
['code', 'odec', 'deco', 'ecod']

```

A generator expression can't use the seq assignment statement of the first generator-function coding, but it *can* achieve the same effect by nesting the `:=` *named-assignment* expression covered in [Chapter 11](#), because `:=` both changes assigned names in the containing scope and returns the assigned value so it appears in the results stream:

```

>>> S = 'code'
>>> G = (S:=(S[1:] + S[:1]) for i in range(len(S)))    # Generator expr, coding 1
>>> list(G)
['odec', 'deco', 'ecod', 'code']

```

Either way, we're still operating on a specific variable here, S. To generalize a generator expression for an arbitrary subject, wrap it in a simple lambda function that takes an argument and returns a generator that uses it (subtle bit: note that seq in the second of these is not local to the generator, but is local to the lambda that encloses it):

```

>>> F = lambda seq: (seq[i:] + seq[:i] for i in range(len(seq)))
>>> list(F(S))
['code', 'odec', 'deco', 'ecod']

>>> F = lambda seq: (seq:=(seq[1:] + seq[:1]) for i in range(len(seq)))
>>> list(F(S))
['odec', 'deco', 'ecod', 'code']

>>> list(F([1, 2, 3]))
[[2, 3, 1], [3, 1, 2], [1, 2, 3]]

>>> F(S)
<generator object <lambda>.<locals>.<genexpr> at 0x100b23300>

```

```
>>> for x in F((1, 2, 3)): print(x, end=' ')
(2, 3, 1) (3, 1, 2) (1, 2, 3)
```

Tester client

Finally, we can use either the generator function or its expression equivalent in [Chapter 18](#)'s `tester` to produce scrambled arguments. As packaged in the module of [Example 20-1](#), the sequence scramblers become reusable tools.

Example 20-1. `scramble.py`

"Generate shuffles of a sequence, by function or expression"

```
def scramble1(seq):
    for i in range(len(seq)):
        yield seq[i:] + seq[:i]      # Yield one shuffle per iteration

scramble2 = lambda seq: (seq[i:] + seq[:i] for i in range(len(seq)))
```

Though it requires a bit of page flipping to see how, moving the values generation out to an external tool like this also makes the testing function noticeably simpler:

```
>>> from scramble import scramble1      # Choose your scrambler
>>> from inter2 import intersect, union # Functions to be tested

>>> def tester(func, items, trace=True):
    for args in scramble1(items):       # Use either generator
        if trace: print(args)
        print(sorted(func(*args)))      # Test for this scramble

>>> tester(intersect, ('aab', 'abcde', 'ababab'), False)
['a', 'b']
['a', 'b']
['a', 'b']
```

To make this work for yourself, make sure all imported files are in the current directory: either copy `inter2.py` from [Chapter 18](#)'s folder to [Chapter 20](#)'s, or copy [Example 20-1](#)'s `scramble.py` to [Chapter 18](#)'s folder and work there (the examples package has already done the former). Alternatively, you can modify the module import search path's `PYTHONPATH` setting to include any folder—as you'll learn when we cover modules in full after the next chapter.

Permutating Sequences

Generators have many other real-world applications—consider parsing attachments in an email message or computing points to be plotted in a GUI. Moreover, other types of sequence scrambles serve central roles in other applications, from searches to mathematics. As is, our sequence scrambler of the prior section is a simple reordering, but some programs warrant the more exhaustive set of all possible orderings we get from *permutations*—produced using recursive functions in both list-builder and generator forms by the module file in [Example 20-2](#).

Example 20-2. *permute.py*

"Permute sequences: as a list or generator of values"

```
def permute1(seq):
    if not seq:
        return [seq]
    else:
        res = []
        for i in range(len(seq)):
            rest = seq[:i] + seq[i+1:]
            for x in permute1(rest):
                res.append(seq[i:i+1] + x)
        return res

def permute2(seq):
    if not seq:
        yield seq
    else:
        for i in range(len(seq)):
            rest = seq[:i] + seq[i+1:]
            for x in permute2(rest):
                yield seq[i:i+1] + x
```

Both of these functions produce the same results, though the second defers much of its work until it is asked for a result. This code is a bit advanced, especially the second of these functions (and to some Python newcomers might even be categorized as cruel and unusual punishment!). Still, as you'll learn in a moment, there are cases where the generator approach can be very useful—and even essential.

Study and test this code for more insight, and add prints to trace if it helps. If it's still a mystery, try to make sense of the first version first; remember that generator functions simply return objects with methods that handle next operations run by for loops at each level, and don't produce any results until iterated; and trace through some of the following examples to see how they're handled by this code.

Permutations produce more orderings than the original scrambler—for N items, we get $N!$ (factorial, not surprise) results instead of just N (24 for $4 * 3 * 2 * 1$). In fact, that's why we need *recursion* here: the number of nested loops is arbitrary, and depends on the length of the sequence permuted. Here's a demo of both shufflers at work:

```
>>> from scramble import scramble1
>>> from permute import permute1, permute2

>>> list(scramble1('abc'))
['abc', 'bca', 'cab']

>>> permute1('abc')
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
>>> list(permute2('abc'))
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']

>>> G = permute2('abc')
>>> next(G)
'abc'
>>> next(G)
'acb'
>>> for x in permute2('abc'): print(x)
...prints six lines...
```


The list and generator versions' results are the same, though the generator minimizes both space usage and delays for results. For larger items, the set of all permutations from both is much larger than the simpler scrambler's:

```
>>> permute1('hack') == list(permute2('hack'))
True
>>> len(list(permute2('hack'))), len(list(scramble1('hack')))
(24, 4)

>>> list(scramble1('hack'))
['hack', 'ackh', 'ckha', 'khac']
>>> list(permute2('hack'))
['hack', 'hac', 'hcak', 'hcka', 'hkac', 'hkca', 'ahck', 'ahkc', 'achk', 'ackh',
 'akhc', 'akch', 'chak', 'chka', 'cahk', 'cakh', 'ckha', 'ckah', 'khac', 'khca',
 'kahc', 'kach', 'kcha', 'kcah']
```

Per [Chapter 19](#), there are nonrecursive alternatives here too, using explicit stacks or queues, and other sequence orderings are common (e.g., fixed-size subsets and combinations that filter out duplicates of differing order), but these require coding extensions we'll forgo here. Experiment further on your own for more insights.

Why generators here: Space, time, and more

Generators are a somewhat advanced tool, and might be better treated as an optional topic, but for the fact that they permeate the Python language today. As we've seen, fundamental built-in tools such as `range`, `map`, dictionary keys, and even files are now generators, so you must be familiar with the concept even if you don't write new generators of your own. Moreover, user-defined generators are increasingly common in Python code that you might come across today—in the Python standard library, for instance.

Though your code is yours to code, the same guidelines given for list comprehensions apply here as well: don't complicate your code with user-defined generators if they are not warranted. Especially for smaller programs and data sets, such tools may not make sense. Simple lists of results may suffice, may be easier to understand, will be garbage-collected automatically, and might be produced quicker (and are today: see the next chapter). Advanced tools like generators that rely on implicit “magic” can be fun to experiment with, but may be subpar when optional.

That being said, there are specific use cases that generators can address well. They can reduce memory footprint in some programs, reduce delays in others, and can occasionally make the impossible possible. Consider, for example, a program that must produce all possible permutations of a nontrivial sequence. Since the number of combinations is a *factorial* that explodes exponentially, the preceding `permute1` recursive list-builder function will either introduce a noticeable and perhaps interminable pause, or fail completely due to memory requirements.

By contrast, the `permute2` recursive generator returns each individual result quickly, and can handle very large result sets. Even at just 10 items, for instance, the difference starts becoming stark (per Python's `math` module):

```
>>> import math
>>> f'{math.factorial(10):,}'          # 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
'3,628,800'

>>> from permute import permute1, permute2
>>> seq = list(range(10))

>>> p1 = permute1(seq)                 # ~17 seconds on a 2.3GHz i9 macOS host
```

```
>>> len(p1)                                # Creates a list of 3.6M numbers
3628800
>>> p1[0], p1[-1]
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

In this case, the `permute1` list builder pauses for 17 seconds to build a 3.6-million-item list, but the *generator* can begin returning individual results immediately—neither `permute2` nor any `next` pause in the following:

```
>>> p2 = permute2(seq)                      # Returns generator immediately
>>> next(p2)                                # Returns each result quickly on demand
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> next(p2)
[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]
```

While collecting all results from the generator is still slow, it's faster than the list builder, and not intended usage:

```
>>> p2 = list(permute2(seq))                 # About 10 seconds, though still impractical
>>> p1 == p2                                # Same set of results generated
True
```

Naturally, we might be able to optimize the list builder's code to run quicker (e.g., an explicit stack instead of recursion might change its performance), but for larger sequences, it's not an option at all—at just 50 items, the number of permutations wholly precludes building a results list, and would take far too long for mere mortals like us in any event (and larger values will overflow the preset recursion stack depth limit: see the preceding chapter). The generator, however, is still viable—despite the factorial size of the problem, it is able to produce individual results immediately:

```
>>> math.factorial(50)
3041409320171337804361260816606476884437764156896051200000000000000

>>> p3 = permute2(list(range(50)))           # permute1 is not an option here!
>>> next(p3)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
44, 45, 46, 47, 48, 49]
```

For more fun—and to yield results that are more variable and less obviously deterministic—we could use Python's `random` module of [Chapter 5](#) to randomly shuffle the sequence to be permuted before the permuter begins its work. In the following, for example, each `permute2` and `next` call returns immediately as before and permutes a random sequence, but `permute1` hangs (and presumably perishes from memory starvation if allowed to run long enough):

```
>>> seq = list(range(20))
>>> print(f'{math.factorial(20):,}')         # permute1 is not an option here
2,432,902,008,176,640,000

>>> import random
>>> random.shuffle(seq)                      # Shuffle sequence randomly first
>>> p = permute2(seq)
>>> next(p)
[10, 19, 5, 6, 2, 13, 1, 8, 11, 7, 14, 16, 4, 3, 0, 18, 9, 12, 17, 15]
>>> next(p)
[10, 19, 5, 6, 2, 13, 1, 8, 11, 7, 14, 16, 4, 3, 0, 18, 9, 12, 15, 17]

>>> random.shuffle(seq)
>>> p = permute2(seq)
>>> next(p)
[17, 15, 14, 7, 10, 8, 2, 6, 18, 19, 13, 4, 1, 12, 5, 0, 3, 9, 16, 11]
```

```
>>> next(p)
[17, 15, 14, 7, 10, 8, 2, 6, 18, 19, 13, 4, 1, 12, 5, 0, 3, 9, 11, 16]
```

In fact, we might be able to use the random shuffler instead of manual shuffles in some roles, as long as we either can assume that it won't repeat shuffles during the time we consume them, or test its results against prior shuffles to avoid repeats—and hope that we do not live in the strange universe where a random sequence repeats the same result an infinite number of times! When full coverage is important, manual shuffles offer better control.

The main point here is that generators can sometimes produce results from large solution sets when list builders cannot. Then again, it's not clear how common such use cases may be in the real world, and this doesn't necessarily justify the *implicit* flavor of value generation that we get with generator functions and expressions. As you'll see in [Part VI](#), value generation can also be coded as iterable objects with *classes*. Class-based iterables can produce items on request too, and are more *explicit* than the magic objects and methods produced for generator functions and expressions.

Part of programming is finding a balance among trade-offs like these, and there are no absolute rules here. While the benefits of generators may sometimes justify their use, maintainability counts too. Like comprehensions, generators also offer an *expressiveness* and *code economy* that's hard to resist if you understand how they work—but you'll want to weigh this against the frustration of coworkers who might not.

Example: Emulating zip and map

Let's take a quick look at one more example of generators in action that illustrates just how expressive they can be. Once you know about comprehensions, generators, and other iteration tools, you'll find that emulating many of Python's functional built-ins is both straightforward and instructive, and can be useful for custom roles.

For example, we've already seen how the built-in `zip` and `map` functions combine iterables and project functions across them, respectively. With multiple iterable arguments, `map` projects the function across items taken from each iterable in much the same way that `zip` combines them, and both functions truncate at the shortest iterable's length:

```
>>> list(zip('abc', 'xyz123'))           # zip combines items from iterables
[('a', 'x'), ('b', 'y'), ('c', 'z')]
>>> list(zip([1, 2, 3], [2, 3, 4, 5]))    # N M-ary iterables: M N-ary tuples
[(1, 2), (2, 3), (3, 4)]
>>> list(zip([-2, -1, 0, 1, 2]))          # 1 5-ary iterable: 5 1-ary tuples
[(-2,), (-1,), (0,), (1,), (2,)]

>>> list(map(abs, [-2, -1, 0, 1, 2]))      # Single iterable: 1-ary function
[2, 1, 0, 1, 2]
>>> list(map(pow, [1, 2, 3], [2, 3, 4, 5])) # N iterables: N-ary function
[1, 8, 81]
```

As covered earlier, both also work on any type of iterable, including files that read their lines automatically. Though they're ultimately used for different purposes, if you study these examples long enough, you might notice a relationship between `zip` results and `map` function arguments that our next example can exploit.

Coding Your Own map

Although the `map` and `zip` built-ins are fast and convenient, they're easy to implement in customizable code of our own. In the preceding chapter, for example, we wrote a function that emulated the `map` built-in for a *single* iterable argument. Per [Example 20-3](#), it doesn't take much more work to allow for *multiple* iterables, as the built-in does.

Example 20-3. *mymap-lists.py*

"Emulate map: support multiple arguments, build a list result"

```
def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        res.append(func(*args))
    return res

print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

This version relies upon `*` argument syntax—it *collects* multiple sequence (really, iterable) arguments; *unpacks* them as `zip` arguments to combine; and then *unpacks* the combined `zip` results as arguments to the passed-in function. That is, we're using the fact that the zipping is essentially a nested operation in mapping. The test code at the bottom applies this to both one and two sequences to test—with the same results generated by the built-in `map`:

```
$ python3 mymap-list.py
[2, 1, 0, 1, 2]
[1, 8, 81]
```

Really, though, the preceding code exhibits the classic *list comprehension pattern*, building a list of operation results within a `for` loop. We can rewrite our mapper more concisely as an equivalent one-line list comprehension:

```
def mymap(func, *seqs):
    return [func(*args) for args in zip(*seqs)]
```

When this is run the result is the same as before, but the code is more concise and might run faster (more on performance in [Chapter 21](#)). Both of the preceding `mymap` versions build result lists all at once, though, and this can waste memory for larger lists. Now that we know about *generator* functions and expressions, it's simple to recode both these alternatives to produce results on demand instead, per [Example 20-4](#).

Example 20-4. *mymap-generate.py*

"Emulate map: support multiple arguments, generate results on request"

```
def mymap_func(func, *seqs):
    for args in zip(*seqs):
        yield func(*args)

def mymap_expr(func, *seqs):
    return (func(*args) for args in zip(*seqs))

for mymap in (mymap_func, mymap_expr):
```

```
print(list(mymap(abs, [-2, -1, 0, 1, 2])))
print(list(mymap(pow, [1, 2, 3], [2, 3, 4, 5])))
```

Both these new versions produce the same results but return generators designed to support the iteration protocol—the first yields one result at a time explicitly, and the second returns a generator expression's result to do the same implicitly. As generators, we must wrap them in `list` calls to force them to produce their values all at once:

```
$ python3 mymap-generate.py
[2, 1, 0, 1, 2]
[1, 8, 81]
[2, 1, 0, 1, 2]
[1, 8, 81]
```

No real work is done here until the `list` calls force the generators to run by activating the iteration protocol. The `list` calls allow for display in testing, but this is not generally desirable in other contexts. The generators returned by these functions themselves, as well as the generators returned by the `zip` built-in they use, produce results only on demand, and that's the *whole point* of generators—delaying results saves memory space, and avoids pausing callers for full results.

Coding Your Own `zip` and 2.X `map`

Of course, much of the secret behind the success of the examples shown so far lies in their use of the `zip` built-in to combine arguments from multiple iterables. Using iteration tools, we can also code workalikes that emulate both today's truncating `zip`, as well as the former padding behavior of Python 2.X's `map` when passed a `None` for its function—a still potentially useful tool despite its demise in 3.X, and the sort of thing that custom code can provide. Per [Example 20-5](#), `zip` and this padding `map` are related in utility and nearly the same in code.

Example 20-5. myfptools-list.py

Emulate zip and padding map: build lists

```
def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    res = []
    while all(seqs):
        res.append(tuple(S.pop(0) for S in seqs))
    return res

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    res = []
    while any(seqs):
        res.append(tuple((S.pop(0) if S else pad) for S in seqs))
    return res

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))
```

Both of these functions work on any type of iterable object because they run their arguments through the `list` built-in to force result generation (e.g., files would work as arguments, in addition to sequences like strings). Notice the use of the `all` and `any` built-ins here—as we saw briefly in [Chapter 14](#), these

return True if all or any items in an iterable are True (or equivalently, nonempty), respectively. These built-ins are used to stop looping when any or all of the listified arguments become empty after deletions.

Also note the use of the *keyword-only* argument, `pad`; unlike the 2.X `map`, our version will allow any `pad` object to be specified. When these functions are run, the following results are printed—a zip and two padding maps:

```
$ python3 myfptools-list.py
[('a', 'x'), ('b', 'y'), ('c', 'z')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2'), (99, '3')]
```

These functions aren't amenable to list comprehension translation because their loops are too specific. As before, though, while our `zip` and `map` workalikes currently build and return result lists, it's just as easy to turn them into *generators* with `yield` so that they each return one piece of their result set at a time. [Example 20-6](#) shows how.

Example 20-6. `myfptools_generate.py`

```
# Emulate zip and padding map: generate results

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    while all(seqs):
        yield tuple(S.pop(0) for S in seqs)

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    while any(seqs):
        yield tuple((S.pop(0) if S else pad) for S in seqs)
```

The results are the same as before, but this file assumes it will be used or tested elsewhere (it has no self-test code), and we need to use `list` again to force the generators to yield their values for display in the REPL:

```
$ python3
>>> from myfptools_generate import myzip, mymapPad
>>> list(myzip('abc', 'xyz123'))
[('a', 'x'), ('b', 'y'), ('c', 'z')]
>>> list(mymapPad('abc', 'xyz123', pad=99))
[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2'), (99, '3')]
```

Finally, here's an alternative implementation of our `zip` and `map` emulators—rather than deleting arguments from lists with the `pop` method, the following versions do their job by calculating the minimum and maximum *argument lengths*. Armed with these lengths, it's easy to code nested list comprehensions to step through argument index ranges:

```
def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return [tuple(S[i] for S in seqs) for i in range(minlen)]

def mymapPad(*seqs, pad=None):
    maxlen = max(len(S) for S in seqs)
    index = range(maxlen)
    return [tuple((S[i] if len(S) > i else pad) for S in seqs) for i in index]
```

Because these use `len` and indexing, they assume that arguments are *sequences* or similar, not arbitrary iterables, much like our earlier sequence scramblers and permuters. The outer comprehensions here step through argument index ranges, and the inner comprehensions (passed to `tuple`) step through the passed-in sequences to pull out arguments in parallel (though not at the same time: see the next section!). When they're run, the results are as before.

Most strikingly, generators and iterators seem to run rampant in this example. The arguments passed to `min` and `max` are generator expressions, which run to completion before the nested comprehensions begin iterating. Moreover, the nested list comprehensions employ two levels of delayed evaluation—the `range` built-in is an iterable, as is the generator expression argument to `tuple`.

In fact, *no* results are produced here until the square brackets of the list comprehensions request values to place in the result list—which forces all the comprehensions and generators to run. To turn these functions *themselves* into generators instead of list builders, simply use parentheses instead of square brackets again. Here's the case for our `zip`:

```
def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return (tuple(S[i] for S in seqs) for i in range(minlen))

S1, S2 = 'abc', 'xyz123'
print(list(myzip(S1, S2)))           # Go!... [('a', 'x'), ('b', 'y'), ('c', 'z')]
```

In this case, it takes a `list` call to activate the generators and other iterables to produce their results. Experiment with these on your own for more details. Developing further coding alternatives is left as a suggested exercise (but see the sidebar “[Why You Will Care: Iteration Versus Python Morph](#)” on page 526 for an exploration of one such option). Here, we have time for just one more related tale from the generations saga.

Asynchronous Functions: The Short Story

Now that you've survived all the foregoing twists and turns of the functions story in Python, there's one last topic to go. Python 3.5 debuted an extension called *asynchronous* (usually shortened to *async*) functions, which are able to manually pause their execution while waiting for a result, in order to allow other such functions to run.

Such tools are available in other languages (e.g., JavaScript), which provided some of the inspiration and blueprint for their appearance in Python. Indeed, `match`, `:=`, f-strings, type hinting, and a host of other Python tools owe their presence to this same programming-languages arms race, whose logical outcome would make all languages the same.

Origins aside, the pathological use case for the `async` extension is input/output (IO) operations: a function can pause itself until an IO transfer completes so that other parts of the program can run during the wait. While other tools like multiple threads can and still do address this need too, some may judge `async` functions to be a lighter-weight option.

That said, there are downsides to this topic. For one thing, it has been morphing almost constantly since its first murmurs in Python 3.4, which makes it difficult to document in books with lifespans much longer than web pages and blogs. For another, it's something of an all-or-nothing proposition, because `async` code requires other `async` code.

More fundamentally, `async` functions are part of the applications-level domain of *parallel programming*—a category that also includes multiprocessing and multithreading. As such, they are a heavyweight subject of interest to only a subset of Python’s user base, and a lot to ask of newcomers to Python or programming in general. Especially given Python’s already skyrocketing complexity, this may be best taken as an optional add-on topic, and out of scope here.

Nevertheless, `async` functions are not truly optional in Python: they were added deeply to the language as new syntax, via the `async` and `await` reserved words, and are now fair game in code you may have to reuse (and job-interview tests you may have to suffer). For all these reasons, this section provides just enough of an overview to whet your appetite, but defers to Python’s online resources for more details when you’re ready to tackle this advanced topic.

Async Basics

In a nutshell, `async` functions enable a *cooperative*, nonpreemptive multitasking model, with coding trade-offs similar to those imposed by asynchronous network servers. For instance, tasks must generally be short-lived to avoid monopolizing the host device’s processor. Though begun earlier, Python 3.5 made this part of the language with the `async def` statement and `await` expression, as well as the `async for` iterator and `async with` context manager.

`Async` tools and syntax can be used only in functions defined with `async def`. Like the `yield` and `yield from` that this model extend, `async def` causes a function to be compiled specially: when called, instead of running its body, such a function returns an *awaitable* object that supports an expected method-based protocol.

To invoke this object, functions `await` results explicitly, or use an `async for` or `async with` that does so implicitly. Moreover, an *event loop* must be launched to run the show at large. Although the event loop runs just one task at a time (they’re not truly parallel), it can switch to other tasks when a task runs an `await` for a pending result.

Running serial tasks with normal blocking calls

As usual, this may be easier to grok in code than narrative, so let’s turn to some examples to demo the salient bits. All of this section’s example snippets live in the file *all_async_demos.py* in the examples package, previewed in [Example 20-7](#). To work along, either cut and paste from this file or emedia, or run the examples file in its entirety or with parts stubbed out with triple quotes. All examples here assume that the preamble at the top of [Example 20-7](#) has been run.

Example 20-7. *all_async_demos.py* (preamble)

```
import time, asyncio
def now():
    return time.strftime('%H:%M:%S')      # Local time, as hour:minute:second

...all examples here...
```

We need a handful of standard-library tools here: some in the `asyncio` standard-library module are required, and we’ll display times with `time.strftime` and pause with `sleep` calls in both `time` and `asyncio` to simulate a long-running task. See Python’s library manual for these tools’ fine print that we’ll skip here for space and scope.

To get started, consider the following *non-async* code, in which a `main` function calls another, *producer*, that simply pauses for two seconds using the `time.sleep` call before returning a result. The `sleep` stands

in for a real blocking operation—like an IO call, network transfer, or user interaction. As usual with normal functions, `main` simply waits for each producer call to run to completion and return a result, before it proceeds with a next step:

```
def producer(label):
    time.sleep(2)                # Pause for two seconds: blocking
    return f'All done, {label}, {now()}'  # And return a result

def main():
    print('Start =>', now())
    print(producer(f'serial task 1'))    # Run three steps in sequence
    print(producer(f'serial task 2'))    # Waiting for each one to finish
    print(producer(f'serial task 3'))    # Before doing anything else
    print('Stop =>', now())

main()
```

When run, this code produces the following mundane results. As you can see, each producer call runs a two-second sleep before another starts, so the total time required is six seconds—two for each call run in series by `main`:

```
Start => [19:00:28]
All done, serial task 1, [19:00:30]
All done, serial task 2, [19:00:32]
All done, serial task 3, [19:00:34]
Stop  => [19:00:34]
```

Running concurrent tasks with “await” and “async def”

Now, let’s do the same work with `async` functions. In the next listing, the producer is coded as an `async def` function, which makes Python compile it specially: it’s now a *coroutine*—an object which, when called, doesn’t produce its result, but instead returns an automatically created object that supports the coroutine method protocol. That protocol generally includes a method named `__await__`, which returns an iterator that produces coroutine results on `__next__`, but may instead use `__anext__` for generators. We can safely ignore all of these here.

The `async def` syntax is always required for functions that use `await`, as well as `async for` and `async with` demoed ahead. Because `async def` is required to *both* use and define awaitable coroutines, it tends to spread viruslike throughout code (in fact, some tools now come with libraries in two separate forms, asynchronous and not).

Also in the following, producer uses `await` to suspend itself until the sleep expires. `await` is similar to a `yield from`, but the event loop is free to pause the coroutine running the `await` and resume another if the awaited result is not ready. Notice that `await` is used as a statement in producer; it’s also an expression that returns the awaited result, as in `main`, where it returns the result of the awaited objects:

```
async def producer(label):
    await asyncio.sleep(2)        # await requires async
    return f'All done, {label}, {now()}'  # Call nonblocking/awaitable sleep
                                     # Result of await expression

async def main():
    print('Start =>', now())
    task1 = asyncio.create_task(producer(f'async task 1'))
    task2 = asyncio.create_task(producer(f'async task 2'))
    task3 = asyncio.create_task(producer(f'async task 3'))
    print(await task1)
    print(await task2)
```

```

print(await task3)                                # Wait for tasks to finish
print('Stop =>', now())

asyncio.run(main())                               # Start event-loop schedule

```

Crucially, the awaited `asyncio.sleep` call is like `time.sleep`, but is both *nonblocking*—it uses the event-loop’s API to ensure that the caller doesn’t pause for its completion—and *awaitable*—it uses `async def` to allow itself to be paused until a timeout elapses. Operations at the end of `await` chains must generally follow this model.

Also crucially, `main` uses `asyncio.create_task` to wrap each producer object in a *task*—an object run by the event loop that suspends execution of the coroutine it wraps while the coroutine waits for completion of a *future*, and returns the result of its wrapped coroutine when awaited itself. Future objects signal progress in ways that are too low level for this overview; here, it’s enough to know that each task suspends itself while its coroutine is sleeping.

As a result, tasks can be overlapped in time, to run *concurrently*. In our code, all three tasks—and the producer coroutines they manage—finish at the same time, and the total program takes just two seconds instead of six:

```

Start => [19:00:34]
All done, async task 1, [19:00:36]
All done, async task 2, [19:00:36]
All done, async task 3, [19:00:36]
Stop  => [19:00:36]

```

Naturally, we can use loops here to make the steps more automated. The following runs the same as the prior version—taking two total seconds, and two for each overlapped task:

```

async def producer(label):
    await asyncio.sleep(2)
    return f'All done, {label}, {now()}'

async def main():
    print('Start =>', now())
    tasks = []
    for i in range(3):
        tasks.append(asyncio.create_task(producer(f'async task {i+1}')))
    for task in tasks:
        print(await task)
    print('Stop at', now())

asyncio.run(main())

```

In both `producer` and `main`, an `await`, much like a `yield from`, suspends execution of the coroutine that runs it until the awaited object returns its result. Together with the event loop and tasks, the net effect interleaves code.

How not to use `async` functions

To understand the preceding example, it may help to see other codings that do *not* work. In this example, the `awaits`, tasks, and event loop are all essential: the code won’t work without them. For one thing, calling `main` directly without routing it to the event loop fails right out of the gate with a warning. The call to `main` simply creates an awaitable object but does nothing with it (for brevity, snippets in this section show just differing code, followed by its results):

```

async def main():
    print(producer('xxx'))

main()

# Result...
.../all_async_blunders.py:12: RuntimeWarning: coroutine 'main' was never awaited

```

For another, calling `producer` directly instead of awaiting it fails too, and for the same reason. Much like generators, calling a coroutine returns the awaitable object, but doesn't run it; we must `await` the awaitable to make it go, just like we have to run `next` to get results from a generator:

```

async def main():
    print(producer('xxx'))

asyncio.run(main())

# Result...
<coroutine object producer at 0x10142e740>
.../all_async_blunders.py:22: RuntimeWarning: coroutine 'producer' was never awaited

```

More vitally, tasks are key to the *concurrency* here. If the example's `main` simply runs an `await` for a `producer` directly, it will pause for the sleep too. In the following miscoding, both coroutines await the sleep's end, and the two producers run serially—yielding a total of four seconds for the program at large instead of two:

```

async def main():
    print('Start =>', now())
    print(await producer('xxx'))
    print(await producer('yyy'))
    print('Stop =>', now())

asyncio.run(main())

# Result...
Start => [18:41:56]
All done, xxx, [18:41:58]
All done, yyy, [18:42:00]
Stop => [18:42:00]

```

This is also true if we defer the awaits by assignments—`main` continues to run after creating the `producer` coroutine awaitables, but still hangs for each later `await` that runs the coroutine's code, with no concurrency to be found:

```

async def main():
    print('Start =>', now())
    p1 = producer('xxx')
    p2 = producer('yyy')
    print('Await =>', now())
    print(await p1)
    print(await p2)
    print('Stop =>', now())

asyncio.run(main())

# Result...
Start => [18:42:00]
Await => [18:42:00]
All done, xxx, [18:42:02]

```

```
All done, yyy, [18:42:04]
Stop => [18:42:04]
```

That's why tasks are needed in the prior example: they enable coroutines to be paused and run by the event loop to overlap their execution. That is, they enable *concurrency*—along with alternatives like those of the next section.

Running concurrent tasks with “as_completed” and “gather”

But enough common mistakes: let's get back to coding concurrent coroutines the right way. Besides creating tasks, we can use the `gather` method to wait for all of our coroutines automatically, or `as_completed` to wait for results as they finish. The latter can be used in a normal `for` to watch for completions:

```
async def producer(label):
    await asyncio.sleep(2)
    return f'All done, {label}, {now()}'

async def main():
    print('Start =>', now())
    coros = [producer(f'async task {i+1}') for i in range(3)]
    for nextdone in asyncio.as_completed(coros):
        print(await nextdone)
    print('Stop at', now())

asyncio.run(main())
```

This also finishes in two total seconds, though coroutine results may filter in in any order (`as_completed` can also be used with `async for` of the next section in ways covered by Python's manuals):

```
Start => [19:00:38]
All done, async task 2, [19:00:40]
All done, async task 3, [19:00:40]
All done, async task 1, [19:00:40]
Stop at [19:00:40]
```

Alternatively, the `gather` call schedules tasks for coroutines, awaits all its arguments, and returns a list of task return values in the same order as its arguments when all have finished:

```
async def producer(label):
    await asyncio.sleep(2)
    return f'All done, {label}, {now()}'

async def main():
    print('Start =>', now())
    coro1 = producer(f'async task 1')
    coro2 = producer(f'async task 2')
    coro3 = producer(f'async task 3')
    results = await asyncio.gather(coro1, coro2, coro3)
    print(results)
    print('Stop at', now())

asyncio.run(main())
```

Results vary because they are in a list (formatted for display here), but it's still just two seconds for the entire gig:

```
Start => [19:00:40]
['All done, async task 1, [19:00:42]',
```

```

    'All done, async task 2, [19:00:42]',
    'All done, async task 3, [19:00:42]'
Stop at [19:00:42]

```

For any line counters in the audience, a starred list comprehension can be used to create awaitables for gather more succinctly, and produces the same two-second result as the preceding version:

```

async def producer(label):
    await asyncio.sleep(2)
    return f'All done, {label}, {now()}'

async def main():
    print('Start =>', now())
    print(await asyncio.gather(*[producer(f'async task {i+1}') for i in range(3)]))
    print('Stop at', now())

asyncio.run(main())

```

While this code works, Python’s manuals today include a note that recommends `TaskGroup` over `gather` on the grounds of better exception handling. Should you care to follow that advice, you’ll need to move on to the next section.

Running concurrent tasks with “`async with`” and “`async for`”

As yet another concurrency option, the combination of the `async with` statement and a manager object can automatically handle both protocols and exceptions, and wait for tasks to finish. As half of this pair, `async with` internally awaits the awaitables returned by its subject’s `__aenter__` and `__aexit__` methods, the same way that the normal `with` statement calls the context-manager protocol’s `__enter__` and `__exit__`.

Having said that, we unfortunately won’t study either the normal `with` or its method protocols until [Part VII](#), because they require knowledge of classes and OOP (yet again, Python’s toolset assumes you must already know Python to use Python!), so you’ll have to take this on faith for now. In the following code, though, it’s enough to know that using a `TaskGroup` asynchronous context manager in concert with `async with` will automatically await results from all of its tasks before the statement exits, as well as handle some thorny issues involving exceptions:

```

async def producer(label):
    await asyncio.sleep(2)
    return f'All done, {label}, {now()}'

async def main():
    print('Start =>', now())
    async with asyncio.TaskGroup() as tg:
        tasks = [tg.create_task(producer(f'async task {i+1}')) for i in range(3)]
    for task in tasks:
        print(task.result())
    print('Stop at', now())

asyncio.run(main())

```

The net effect runs all three producer calls concurrently in two seconds as before:

```

Start => [19:00:44]
All done, async task 1, [19:00:46]
All done, async task 2, [19:00:46]
All done, async task 3, [19:00:46]
Stop at [19:00:46]

```

Finally, when a function is coded with `async def` and also uses `yield`, it's considered an *asynchronous generator*. When called, it returns an asynchronous iterable object which uses the `__anext__` method noted earlier, and can be used in an `async for` statement to execute the body of the function. (And no, there are no `async` variants of other statements like `while` or `if`; `with` and `for` were the only targets deemed worthy of the `async` twist—so far.)

Internally, `async for` uses methods `__aiter__` and `__anext__` much like the normal iteration protocol's `__iter__` and `__next__` that we studied in [Chapter 14](#). The `__anext__` method returns an awaitable that produces a next value, and raises `StopAsyncIteration` to signal the end of values, instead of `StopIteration`. `async for` simply gets a next value by awaiting a call to `__anext__`, and running the loop's block of code for every value obtained this way. This encroaches on classes and OOP again, but statement users may ignore the details.

To avoid confusion, keep in mind that `async for` awaits the *next* item in the iterable it scans—which is not the same as awaiting for something else in the loop body of a normal `for`, as in prior examples here. Moreover, `async for` does not automatically *parallelize* anything—it's really just like a normal `for`, but adds an `await` for each next item obtained from its iterable, instead of issuing a potentially blocking call. `async for`'s implicit awaits between loops allow it to be interleaved with other ready-to-run tasks, but don't help much otherwise:

```
async def producer(label):
    for i in range(3):
        await asyncio.sleep(2)
        yield f'All done, {label} {i+1}, {now()}'

async def main():
    print('Start =>', now())
    async for reply in producer('async task'):
        print(reply)
    print('Stop at', now())

asyncio.run(main())
```

As coded here, the result is the same six-second showing of the original serial version, because there's nothing else to run in this simple demo during the sleeps:

```
Start => [19:00:46]
All done, async task 1, [19:00:48]
All done, async task 2, [19:00:50]
All done, async task 3, [19:00:52]
Stop at [19:00:52]
```

And lest this section hasn't yet managed to send you screaming into the night, it's also possible to use `await` and `async for` within a *comprehension* that's coded in an `async def`—but this must remain a story for another day:

```
async def hmm():
    result = [i async for i in asynciter() if i > 0]      # This code is not real
    result = [await func() for func in coroutines]        # It's also unreal...
```

The Async Wrap-Up

That's all the time and space we have for this topic (humanely, perhaps). Python's `async` functions may be more useful than type hinting (which, as we saw in [Chapter 6](#), is both completely unused and at odds with Python's core ideas), but a full survey of either topic could fill chapters or books, and would be of interest to only a subset of Python users.

Moreover, this text isn't covering Python's multithreading or multiprocessing tools at all, even though they are at least as valid as `async` functions for parallelizing tasks. The fact that `async` alone was afforded dedicated syntax in the language both subjectively favors just one option in this domain and raises the bar for Python newcomers.

That said, your goals and views may differ, so please see Python's standard manuals for the full story if and when you wish to wade deeper into the `async` sea. Here, we must step back from the ledge of optional and stunningly convoluted extensions, and get back to the fundamentals that nearly all Python programs and programmers use.

Chapter Summary

This chapter wrapped up our coverage of built-in comprehension and iteration tools. It explored list comprehensions in the context of functional tools, and presented generator functions and expressions as additional iteration protocol tools. As a finale, we also explored some larger examples of iterations, comprehensions, and generations in action, and briefly glimpsed the asynchronous-functions extension to pique further study. Though we've now seen all the built-in iteration tools, the subject will resurface when we explore user-defined iterable classes in [Chapter 30](#).

The next chapter is something of a continuation of the theme of this one—it rounds out this part of the book with a case study that times the performance of the tools we've studied here, and serves as a more realistic example at the midpoint of this book (yes, you're half done!). Before we move ahead to benchmarking comprehensions and generators, though, this chapter's quizzes give you a chance to review what you've learned about them here.

Test Your Knowledge: Quiz

1. What is the difference between enclosing a list comprehension in square brackets and parentheses?
2. How are generators and iterators related?
3. How can you tell if a function is a generator function?
4. What does a `yield` statement do?
5. How are `map` calls and list comprehensions related? Compare and contrast the two.
6. What do `async def` and `await` mean in a Python script?

Test Your Knowledge: Answers

1. List comprehensions in square brackets produce the result list all at once in memory. When they are enclosed in parentheses instead, they are actually generator expressions—they have the same internal syntax and a similar meaning but do not produce the result list all at once. Instead, generator expressions return a generator object, which yields one item in the result at a time when used in an iteration tool or iterated manually with `next`.
2. Generators are iterable objects that support the iteration protocol automatically—they have an iterator with a `__next__` method (run by `next`) that repeatedly advances to the next item in a series of results and raises an exception at the end of the series. In Python, we can code generator *functions* with `def` and `yield`, generator *expressions* with parenthesized comprehensions, and generator objects with *classes* that define a special method named `__iter__` (discussed later in the book).

3. A generator function has a `yield` statement (with or without its `from` extension) somewhere in its code. Generator functions are otherwise identical to normal functions syntactically, but they are compiled specially to return an iterable generator object when called. That object retains state and code location between values. (This means that deleting `yield` makes a function normal, but code deletions can cause all sorts of issues.)
4. When present, this statement makes Python compile the function specially as a generator; when called, the function returns a generator object that supports the iteration protocol. When the `yield` statement is run, it sends a result back to the caller and suspends the function's state; the function can then be resumed after the last `yield` statement, in response to a `next` built-in or `__next__` method call issued by the caller. In more advanced roles, the generator `send` method similarly resumes the generator, but can also pass a value that shows up as the `yield` expression's value. Generator functions may also have a `return` statement, which terminates the generator (and attaches an optional value to the automatic `StopIteration` exception).
5. The `map` call is similar to a list comprehension—both produce a series of values, by collecting the results of applying an operation to each item in a sequence or other iterable, one item at a time. The primary difference is that `map` applies a *function* call to each item, and list comprehensions apply arbitrary *expressions*. Because of this, list comprehensions are more general; they can apply a function call expression like `map`, but `map` requires a function to apply other kinds of expressions. List comprehensions also support extended syntax—nested `for` loops, and `if` clauses that subsume the `filter` built-in. `map` also differs by producing a *generator* of values; the list comprehension materializes the result list in memory all at once, and this may matter for large lists.
6. The `async def` is used to define an asynchronous function (usually called a *coroutine*), and `await` waits for another asynchronous object to produce its value, possibly yielding control back to an event loop to allow other code to run during the wait. `async def` is required to use `await`, `async for`, and `async with`, and both an event loop and scheduling calls in module `asyncio` are generally required to achieve concurrency in this model. There's more to the `async` story than told here; see Python's manuals for next steps on this topic, as well as its multithreading and multiprocessing alternatives in Python's standard library.

Why You Will Care: Iteration Versus Python Morph

In [Chapter 14](#), we noted that some built-ins (like `map`) support only a single traversal and are empty after it occurs, and this book promised to show you an example of why that can be important in practice. Now that we've studied iteration topics in full, it can make good on this promise—and demo the negative impacts of Python changes on your code at the same time.

In earlier editions of this book, the following clever coding for `zip` emulation, adapted from a version in Python's docs at the time, worked because `map` returned a physical list instead of a generator of values:

```
def myzip(*args):                                # Before 3.0...
    iters = map(iter, args)
    while iters:                                  # Guarantee >=1, force looping
        res = [next(i) for i in iters]            # Any empty? StopIteration=return
        yield tuple(res)                          # Else return res, suspend state
                                                # Exit=return=StopIteration

>>> list(myzip('ab', 'lmn'))
KeyboardInterrupt
```


Except this broke in Python 3.0. This code relied on that fact that `map`'s result supported multiple scans. When it mutated into a *single-scan* generator in 3.0, as soon as the list comprehension traversed `iters` once, the generator was exhausted but still `True`; later list comprehensions returned `[]`; and this code would loop until killed by a `Ctrl+C`. To fix, the prior edition added a `list` around `map` to make it iterable:

```
def myzip(*args):                                # Before 3.7...
    iters = list(map(iter, args))                 # <== Make iters reiterable, 3.0+
    while iters:                                   # map() is a one-scan generator
        res = [next(i) for i in iters]
        yield tuple(res)

>>> list(myzip('ab', 'lmn'))
RuntimeError: generator raised StopIteration
```

Except this broke in Python 3.7. This code relied on the fact that an uncaught `StopIteration` from any iterable in `iters` was propagated through this function and sufficed to terminate its generation of values. Python 3.7 changed to suppress this “bubbling” propagation of `StopIteration` through a generator and issue a `RuntimeError` in response. The workaround is to explicitly catch the `StopIteration` and return—even though `return` simply raises `StopIteration`:

```
def myzip(*args):                                # After 3.7, and before ?...
    iters = list(map(iter, args))
    while iters:
        try:                                       # <== Catch StopIteration, 3.7+
            res = [next(i) for i in iters]        # StopIteration won't propagate
        except StopIteration:
            return                                # How generators should exit
        yield tuple(res)                          # But exit=return=StopIteration!

>>> list(myzip('ab', 'lmn'))
[('a', 'l'), ('b', 'm')]
```

You'll have to wait until [Part VII](#) for more on the `try` statement, and may have to similarly translate exceptions to returns this way in rare generators that rely on the former bubbling behavior. And while this final version works today, we're not placing any bets on the future.

The takeaways here: wrapping `map` calls in `list` calls is not just for display; generators no longer propagate exit exceptions from code they run; and Python mods can and do break code—both subtly and repeatedly!

The Benchmarking Interlude

Now that we’ve fully explored function coding and iteration tools, we’re going to take a short side trip to put both of them to work. This chapter closes out the function part of this book with a larger case study that times the relative performance of the iteration tools we’ve met so far, in both standard Python and one of its alternatives.

Along the way, this case study surveys Python’s code-timing tools, discusses benchmarking techniques in general, and develops code that’s more realistic and useful than most of what we’ve seen up to this point. We’ll also measure the speed of code we’ve used—data points that may or may not be significant, depending on your programs’ goals.

Finally, because this is the last chapter in this part of the book, we’ll close with the usual sets of “gotchas” and exercises to help you start coding the ideas you’ve read about. First, though, let’s have some fun with tangible Python code.

Benchmarking with Homegrown Tools

We’ve met quite a few iteration alternatives in this book. Like much in programming, they represent trade-offs—in terms of both subjective factors like expressiveness, and more objective criteria such as performance. Part of your job as a programmer and engineer is selecting tools based on factors like these.

In terms of performance, this book has mentioned a few times that list comprehensions and `map` calls sometimes have a speed advantage over `for` loop statements. It has also noted that sorting speed varies with ordering, and the generator functions and expressions of the preceding chapter tend to be slower than all the others, though they minimize memory space requirements and don’t delay the caller for result generation when there are many results to generate.

All that is generally true today in common usage. That being said, benchmarking comes with some big caveats: both code structure and host architecture can influence speed arbitrarily; Python’s performance can vary over time because its internals are constantly being changed and optimized; and the speed of alternative Pythons may differ widely. As noted in [Chapter 2](#), for example, the standard *CPython* may adopt a standard JIT in the future which could change its speed in some contexts, and optimized Pythons like *PyPy* have very different performance profiles.

In short, if you want to verify speed for yourself, you need to time your code, on your own device, with the Python or Pythons you plan to use, and in the here and now. That's a lot of qualifiers, but benchmarking is an empirical task.

Timer Module: Take 1

Luckily, Python makes it easy to time code with custom tools—though perhaps deceptively so. For example, to get the total time taken to run multiple calls to a function with arbitrary positional arguments, [Example 21-1](#)'s function coded in a module file might suffice as a first cut.

Example 21-1. timer0.py

"Simplistic timing function"

```
import time
def timer(func, *args):                # Any positional arguments (only)
    start = time.perf_counter()
    for i in range(100_000):           # Hardcoded reps, range() timed
        func(*args)
    return time.perf_counter() - start  # Total elapsed time in seconds
```

This function fetches time values from Python's standard-library `time` module, and subtracts the system start time from the stop time after running 100,000 calls to the passed-in function with the passed-in arguments.

Time comes from `time.perf_counter`, which is defined to be a portable clock with the best resolution to measure a short duration. The difference between two calls is generally used for performance measurement, but includes time for sleeps and is system-wide time. The alternative `time.process_time` is per-process CPU time, and may also be useful in some roles. See Python's manuals for more details; these calls replace the former and less portable `time.clock`, which was deprecated and dropped since this book's prior edition (breaking most examples here!).

Apart from its `time` calls, the code in [Example 21-1](#) is straightforward and uses tools we've already met. When used on this book's main development computer using macOS and CPython 3.12, its results are these in a REPL:

```
>>> from timer0 import timer
>>> timer(pow, 2, 1000)                # Time to call pow(2, 1000) 100k times
0.08591239107772708
>>> timer(str.upper, 'hack' * 100)     # Time to call 'hack...'.upper() 100k times
0.04990812297910452
```

Though functional and simple, the preceding timer is also fairly limited, and deliberately exhibits some classic mistakes in both function design and benchmarking. Among these, it:

- Hardcodes the *repetitions* count at 100k
- Charges the cost of `range` to the tested function's time
- Doesn't support *keyword* arguments in the tested function
- Doesn't give callers a way to *verify* that the tested function actually worked
- Only gives *total* time, which might fluctuate on some busy host machines

In other words, timing code is more complex than you might expect!

Timer Module: Take 2

To be more general and accurate, let's rewrite the preceding section's code to define still simple but more useful timer utility functions we can both use to see how iteration alternative options stack up now, and apply to other timing needs in the future. These functions, in [Example 21-2](#), are coded in a module file again so they can be used in a variety of programs, and have docstrings giving some basic usage details that help and PyDoc can display; see [Chapter 15](#) for tips on viewing the in-code documentation of this chapter's timing modules.

Example 21-2. timer.py

```
"""
Homegrown timing tools for arbitrary function calls.
Times one call, total of N, best of N, and best of totals of N.
Pass any number of positional and keyword arguments for each func.
"""

import time
timer = time.perf_counter                # See also time.process_time()

def once(func, *pargs, **kargs):         # Collect arguments for func
    """
    Time to run func(...) one time.
    Returns (time, result).
    """
    start = timer()
    result = func(*pargs, **kargs)       # Unpack arguments for func
    elapsed = timer() - start
    return (elapsed, result)             # Return result to verify

def total(reps, func, *pargs, **kargs):  # Collect arguments for func
    """
    Total time to run func(...) reps times.
    Returns (total-time, last-result).
    """
    total = 0                            # Don't charge range() time
    for i in range(reps):
        time, result = once(func, *pargs, **kargs)
        total += time
    return (total, result)               # Return last result to verify

def bestof(reps, func, *pargs, **kargs):
    """
    Best time among reps runs of func(...).
    Returns (best-time, best-time-result).
    """
    return min(once(func, *pargs, **kargs) for i in range(reps))

def bestoftotal(reps1, reps2, func, *pargs, **kargs):
    """
    Best total time among reps1 runs of [reps2 runs of func(...)].
    Returns (best-total-time, best-total-time-last-result).
    """
    return min(total(reps2, func, *pargs, **kargs) for i in range(reps1))
```

Operationally, this module implements both *total* and *best* times, and a *best of totals* that combines the other two. In each mode, it times calls to any subject function that takes any positional and keyword

arguments, by fetching the start time, calling the function with arbitrary arguments, and subtracting the start time from the stop time. Here are the salient points to notice about how this version addresses the shortcomings of its predecessor:

- The *repetitions* count is no longer hardcoded, but passed in as a required argument (or arguments) before the test function and its own arguments, to allow repetitions to vary per call.
- The *range* call's construction and iteration costs are no longer charged to timed functions, because timing has been factored out to the separate *once* function that times just the subject function.
- Any number of both positional and *keyword* arguments for the timed function are now collected and unpacked with starred-argument syntax. They must be sent individually, not in a sequence or dictionary, though callers can unpack argument collections into individual arguments with stars in the top-level call.
- All functions in this module return one of the timed function's *return values* so callers can verify that the function worked. The return value is provided in a two-item result tuple, along with the requested time result.
- New best-of modes return *minimum* times to address fluctuations on the host device, per the next paragraph.

This version's functions also support multiple use cases: *once* times a single call for simple cases; *total* runs many calls, to allow time to accumulate for short-lived functions; *bestof* returns the minimum time among all single calls, to filter out the impacts of other activity on the host; and *bestoftotal* selects the minimum of nested total-time tests, to both apply a best-of filter and run many calls for functions too fast to produce meaningful times.

Importantly, the *min* calls in the best-of variants work to select the best and lowest time, because Python compares collections recursively (from left to right) as we've learned, and result tuples begin with *time*: because it's first in these (*time*, *result*) tuples, time dominates and determines the *min* calls' results:

```
>>> min(tup for tup in [(2.0, 3), (3.0, 3), (1.0, 3), (0.0, 3), (4.0, 3)])
(0.0, 3)
```

From a larger perspective, because these functions are coded in a module file, they become generally useful tools anywhere we wish to import them. Modules and imports were introduced in [Chapter 3](#), and you'll learn more about them in the next part of this book; for now, simply import the module and call its function to use one of this file's timers. Its results on the same host and in a REPL are similar to its *timer0.py* predecessor, but are more robust:

```
>>> import timer                                # Import file in this directory
>>> help(timer)                                  # Display module's docs nicely
>>> reps, text = 100_000, 'hack' * 100

>>> timer.once(pow, 2, 1000)[0]                  # Not useful for fast calls
6.182119250297546e-06
>>> timer.once(str.upper, text)                  # (time, result)
(3.363005816936493e-06, 'HACKHACKHACK...etc...')

>>> timer.total(reps, pow, 2, 1000)[0]            # Compare to timer0 results
0.08979405369609594
>>> timer.total(reps, str.upper, text)[0]        # (time, last call's result)
0.050884191412478685

>>> timer.bestof(50, pow, 2, 1000)[0]           # Not useful for fast calls
1.6265548765659332e-06
```

```
>>> timer.bestof(50, str.upper, text)[0]           # (best time, best time result)
8.619390428066254e-07

>>> timer.bestoftotal(50, reps, pow, 2, 1000)[0]
0.07521858718246222
>>> timer.bestoftotal(50, reps, str.upper, text)[0]
0.03947464330121875
```

The last two calls here calculate the *best-of-totals* times—the lowest time among 50 runs, each of which computes the total time to call a function 100k times—roughly corresponding to the `total` times earlier in this listing, but repeated for a minimum that filters out host fluctuations (and sans an extra charge for range). The function used in these last two calls is really just a convenience that wraps `total` for better accuracy, but this is a common timing mode.

Note that `bestoftotal` might replace its `min` of `total` with code like the following to nest `total` in `bestof`:

```
>>> timer.bestof(50, timer.total, reps, str.upper, text)
(0.07037258706986904, (0.039281503297388554, 'HACKHACKHACK...etc...'))
```

But this isn't quite the same. For one thing, the result is nested tuples, reflecting the nested calls. For another, this really times the entire `total` function, not just the subject function it runs. The net effect charges an extra admin-code overhead that's enough to skew the best-of time up, and make it larger than the best total time shown in the nested tuple. By using `min` of `total` instead, `bestoftotal` avoids timing this overhead skew. Subtle but true!

Timing Runner and Script

Now, to time iteration tool speed (our original goal), we'll write a script that defines and submits test functions to the `timer` module. To make it easy to code a variety of tests, let's first define a utility module that does the heavy lifting as a reusable intermediary. [Example 21-3](#) defines a function named `runner` that takes any number of test functions as arguments and passes them off for timing to the `bestoftotal` function imported from [Example 21-2](#).

Example 21-3. `timer_runner.py`

"Run passed-in test functions with the `timer.py` module"

```
import timer, sys

def runner(*tests):
    results = []
    print('Python', sys.version.split()[0], 'on', sys.platform)

    # Time
    for test in tests:
        besttime, result = timer.bestoftotal(10, 1000, test)
        results.append(result)
        print(f'{test.__name__:<9}: '
              f'{besttime:.5f} => [{result[0]}...{result[-1]}]')

    # Verify
    print('Results differ!'
          if any(result != results[0] for result in results[1:])
          else 'All results same.')
```

Fine points here: this module's runner function displays context with `sys` tools documented in Python's manuals, and steps through all the passed-in functions, printing the `__name__` of each (as we've seen, this is a built-in attribute that gives a function's name). The test-runner code also saves results to verify that they are all the same in a ternary expression at the very end of the process, to be sure we're comparing apples to apples.

Last but not least, the script in [Example 21-4](#) defines the actual tests to be timed and passes them to the imported runner of [Example 21-3](#), which hands them off to the imported timer of [Example 21-2](#). We'll run the file in [Example 21-4](#) as a top-level script to time the relative speeds of the various iteration codings we've studied in this book so far.

Example 21-4. timer_tests.py

```
"Test the relative speed of iteration coding alternatives."

from timer_runner import runner
repslist = list(range(10_000))

def forLoop():
    res = []
    for x in repslist:
        res.append(abs(x))
    return res

def listComp():
    return [abs(x) for x in repslist]

def mapCall():
    return list(map(abs, repslist))           # Use list() to force results

def genExpr():
    return list(abs(x) for x in repslist)    # Use list() to force results

def genFunc():
    def gen():
        for x in repslist:
            yield abs(x)
    return list(gen())                      # Use list() to force results

runner(forLoop, listComp, mapCall, genExpr, genFunc)
```

This script tests five alternative ways to build lists of results. In combination with the test runner, its reported times reflect some 100 million steps for each of the 5 test functions—each builds a list of 10,000 items 1,000 times, and this process is repeated 10 times to get the best-of times. Applying this for each of the 5 test functions yields a whopping 500 million total steps for the script at large (impressive but reasonable on most machines these days).

Notice how we have to run the results of the generator expression and function through the built-in `list` call to force them to yield all of their values; if we did not, we would just produce generators that never do any real work. We must do the same for the `map` result, since it is an iterable, on-demand object as well.

For similar reasons, the inner loops' `range` result is hoisted out to the top of the module to remove its construction cost from total time, and wrapped in a `list` call so that its traversal cost isn't skewed by being a generator. This may be overshadowed by the cost of the inner iterations' loops, but it's best to

remove as many variables as we can. For example, though `range` supports multiple scans, tests' times inflate by some 25% if its `list` wrapper is removed.

Iteration Results

When the top-level script of the prior section is run under the standard CPython 3.12 on the same macOS host, it prints the following with total times in seconds—after cueing the *drum roll*, that is:

```
$ python3 timer_tests.py
Python 3.12.2 on darwin
forLoop : 0.26035 => [0...9999]
listComp : 0.20781 => [0...9999]
mapCall : 0.14399 => [0...9999]
genExpr : 0.41133 => [0...9999]
genFunc : 0.41203 => [0...9999]
All results same.
```

In short, `map` calls are faster than list comprehensions, which are quicker than `for` loops, and both generator expressions and functions come in last and roughly tied for slowest. Perhaps surprisingly, generator expressions run much slower than equivalent list comprehensions today. As warned in the prior chapter, although wrapping a generator expression in a `list` call makes it *functionally* equivalent to a list comprehension, the internal *implementations* of the two expressions appear to differ:

```
return [abs(x) for x in repslist]      # 0.20 seconds
return list(abs(x) for x in repslist)  # 0.41 seconds: differs internally
```

We're also effectively timing the `list` call for the generator test, but given that this does not seem to hamper `map`, it's likely moot. Though the exact cause for the difference would require deeper analysis (and probably source code spelunking), this seems to make sense given that the generator expression must do extra work to save and restore its state during value production; the list comprehension does not, and runs quicker here and in other tests ahead.

Other Python's results

For comparison, following are the same tests' speed results on the same host using the current PyPy—the optimized Python implementation discussed in [Chapter 2](#), whose current 7.3 release implements the Python 3.10 language. PyPy is roughly 5X quicker than CPython here (and up to 10X), though its timing results can vary widely if its JIT has not yet compiled code in full (and a future and currently hypothetical CPython JIT may or may not even the race):

```
$ pypy3 timer_tests.py
Python 3.10.14 on darwin
forLoop : 0.04329 => [0...9999]
listComp : 0.01876 => [0...9999]
mapCall : 0.04132 => [0...9999]
genExpr : 0.08744 => [0...9999]
genFunc : 0.08726 => [0...9999]
All results same.
```

On PyPy alone, list comprehensions win the title in this test today, but generators still lose soundly, and the fact that all of PyPy's results are so much quicker today seems the larger point here. On CPython, `map` is still quickest so far.

Interestingly, these results are very different than they were in this book's prior edition on Windows under CPython 3.3—when generators placed in the middle of the pack between list comprehensions

and for loops, for loops fared substantially worse than they do today, and the script had a different name for historical reasons:

```
C:\code> c:\python33\python timeseqs.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
forLoop : 1.33290 => [0...9999]
listComp : 0.69658 => [0...9999]
mapCall : 0.56483 => [0...9999]
genExpr : 1.08457 => [0...9999]
genFunc : 1.07623 => [0...9999]
```

While these absolute times naturally reflect older and slower test hosts, these tools' *relative* performance has clearly changed in the last 12 years—and probably should be expected to do so again in another dozen!

For more good times: Function calls and map

All of the foregoing is true as advertised, but watch what happens when [Example 21-5](#) performs an *inline* operation on each iteration, such as a + expression, instead of calling a built-in function like `abs`. Only the test functions' operations (in bold) need to be modified here, and the imported and reused runner handles tests generically.

Example 21-5. `timer_tests2.py`

```
from timer_runner import runner
replist = list(range(10_000))

def forLoop():
    res = []
    for x in replist:
        res.append(x + 10)
    return res

def listComp():
    return [x + 10 for x in replist]

def mapCall():
    return list(map((lambda x: x + 10), replist))

def genExpr():
    return list(x + 10 for x in replist)

def genFunc():
    def gen():
        for x in replist:
            yield x + 10
    return list(gen())

runner(forLoop, listComp, mapCall, genExpr, genFunc)
```

Now, `map` is *slower* than the for loop statements, despite the fact that the looping-statements version is larger in terms of code. This could either mean that the need to *call* a user-defined function makes `map` slower—or equivalently, that the *lack* of function calls makes the others quicker. On CPython 3.12 and the same host as before:

```
$ python3 timer_tests2.py
Python 3.12.2 on darwin
forLoop : 0.28682 => [10...10009]
listComp : 0.24389 => [10...10009]
mapCall : 0.49622 => [10...10009]
genExpr : 0.44047 => [10...10009]
genFunc : 0.44476 => [10...10009]
All results same.
```

These results have also been consistent in CPython: the prior edition's Python 3.3 results on a slower machine were again relatively similar, discounting test machine differences. Because the interpreter optimizes so much internally, performance analysis of Python code like this is a very tricky affair. Without numbers, it's virtually impossible to guess which method will perform the best; again, the best you can do is time your code with your test parameters.

In this case, what we can say is that on this Python, using a user-defined `lambda` function in `map` calls seems to slow its performance disproportionately (though `+` is also slower than a trivial `abs` across the board), and that list comprehensions run quickest in this case (though slower than `map` in some others). List comprehensions seem consistently faster than `for` loops, but even this must be qualified—the list comprehension's relative speed might be affected by its extra syntax (e.g., `if` filters), Python changes, and usage modes we did not time here.

For deeper truth, [Example 21-6](#) codes one last takeoff on our tests to apply a simple user-defined function in *all five* iterations timed. Again, we must only modify the relevant (and bold) bits of the test functions themselves.

Example 21-6. timer_tests3.py

```
from timer_runner import runner
repslist = list(range(10_000))

def F(x): return x

def forLoop():
    res = []
    for x in repslist:
        res.append(F(x))
    return res

def listComp():
    return [F(x) for x in repslist]

def mapCall():
    return list(map(F, repslist))

def genExpr():
    return list(F(x) for x in repslist)

def genFunc():
    def gen():
        for x in repslist:
            yield F(x)
    return list(gen())

runner(forLoop, listComp, mapCall, genExpr, genFunc)
```

When coded this way and run in CPython 3.12 on the same host again, `map` improves its relative times, and comes in second between list comprehensions and for loops—instead of being slower than all others as it was for `+`:

```
$ python3 timer_tests3.py
Python 3.12.2 on darwin
forLoop : 0.36206 => [0...9999]
listComp : 0.31181 => [0...9999]
mapCall : 0.35479 => [0...9999]
genExpr : 0.50531 => [0...9999]
genFunc : 0.50290 => [0...9999]
All results same.
```

That is, `map` may be slower simply *because it requires function calls*, and function calls are relatively slow in general. Since `map` can't avoid calling functions, it may lose by association, and the other iteration tools may win when they can use expressions instead. On the other hand, this hypothesis alone can't explain the better showing for `map` in the abs results of the first *timer_tests.py*: calling *built-in* functions may be a special and fast case for `map` in CPython (a theory supported by “[Conclusion: Comparing tools](#)” on [page 548](#) and `ord` at the end of this chapter's benchmarking safari).

All this being said, performance should not be your primary concern when writing Python code—the first thing you should do to optimize Python code is to *not optimize Python code!* Write for readability and simplicity first, then optimize later, if and only if needed. It could very well be that any of the five iteration alternatives we've timed is quick enough for the data sets your program needs to process; if so, program clarity should be the chief goal.

More Module Mods

Our *timer.py* module works as designed, but it could be a bit more user-friendly. Most obviously, its functions require passing in repetition counts as first arguments, and provide no defaults for them—a minor point, perhaps, but less than ideal in a general-purpose tool. To do better, it could allow repetition counts to be passed in as *keyword* arguments with *defaults*, much the same way we did for the print emulators of [Chapter 18](#).

Here, though, these arguments' names would have to be distinct to avoid clashing with those of the timed function (e.g., `_reps` instead of `reps`). While we're at it, the function object could also be a *positional-only* argument to prevent name `func` from clashing with a keyword argument of the same name in the timed subject. [Example 21-7](#) codes the required mods (sans docs for space). Review [Chapter 18](#)'s argument-ordering coverage for more insight.

Example 21-7. *timer2.py*

"Use keyword-only arguments with defaults for `reps`, and positional-only for `func`."

```
import time
timer = time.perf_counter

def once(func, /, *pargs, **kargs):
    start = timer()
    result = func(*pargs, **kargs)
    elapsed = timer() - start
    return (elapsed, result)

def total(func, /, *pargs, _reps=100_000, **kargs):
    total = 0
```

```

    for i in range(_reps):
        time, result = once(func, *pargs, **kargs)
        total += time
    return (total, result)

def bestof(func, /, *pargs, _reps=5, **kargs):
    return min(once(func, *pargs, **kargs) for i in range(_reps))

def bestoftotal(func, /, *pargs, _reps1=50, **kargs):
    return min(total(func, *pargs, **kargs) for i in range(_reps1))    # _reps => **

```

This version is not backward compatible: it uses different names and modes for repetition arguments, which means the *timer_runner.py* we wrote earlier would require a minor edit to use it (see the examples package's *timer2.py*). Otherwise, it works the same—compare its output on the same host with *timer.py*'s results listed at [Example 21-2](#):

```

$ python3
>>> import timer2
>>> timer2.total(pow, 2, 1000, _reps=100_000)[0]
0.0865794476121664
>>> timer2.total(str.upper, 'hack' * 100, _reps=100_000)[0]
0.04620114527642727

>>> timer2.bestoftotal(pow, 2, 1000, _reps1=50, _reps=100_000)[0]
0.07179990829899907
>>> timer2.bestoftotal(str.upper, 'hack' * 100, _reps1=50, _reps=100_000)[0]
0.0393348871730268

```

This time, though, we can allow the functions' repetition defaults to apply or not:

```

>>> timer2.total(str.upper, 'hack' * 100)[0]
0.047992002684623
>>> timer2.bestoftotal(str.upper, 'hack' * 100)[0]
0.03935634717345238
>>> timer2.bestoftotal(str.upper, 'hack' * 100, _reps=10_000)[0]
0.00393257150426507

```

Notice how the `_reps` argument for `total`, if passed, in `bestoftotal` calls is propagated along in `**kargs` because it's not matched otherwise. For more vetting of this, time user-defined functions with richer argument headers as in the following. Per the first four timings, passing keyword arguments to a timed function adds a small time cost for unpacking—an unavoidable overhead shared by the original *timer.py*, but irrelevant when collecting relative times:

```

>>> def f(a, b, c=88, d=99): return(a, b, c, d)

>>> f(1, 2)
(1, 2, 88, 99)
>>> timer2.bestoftotal(f, 1, 2, _reps1=50, _reps=100_000)
(0.014080120716243982, (1, 2, 88, 99))
>>> timer2.bestoftotal(f, 1, 2, c=66, d=77, _reps1=50, _reps=100_000)
(0.021575820166617632, (1, 2, 66, 77))

>>> timer2.bestoftotal(f, 1, 2, c=66, d=77, _reps1=50)
(0.021694606635719538, (1, 2, 66, 77))
>>> timer2.bestoftotal(f, 1, 2, c=66, d=77, _reps=100_000)
(0.021556788589805365, (1, 2, 66, 77))

>>> def f(a, *b, c=88, **d): return(a, b, c, d)

>>> f(1, 2, 3, c=66, d=77, e=88)

```

```
(1, (2, 3), 66, {'d': 77, 'e': 88})
>>> timer2.bestoftotal(f, 1, 2, 3, c=66, d=77, e=88)
(0.030859854072332382, (1, (2, 3), 66, {'d': 77, 'e': 88}))

>>> timer2.bestoftotal(f, 1, 2, 3, c=66, d=77, e=88, _reps1=10)
(0.030802161898463964, (1, (2, 3), 66, {'d': 77, 'e': 88}))
>>> timer2.bestoftotal(f, 1, 2, 3, c=66, d=77, e=88, _reps=10_000)
(0.0030745575204491615, (1, (2, 3), 66, {'d': 77, 'e': 88}))
```

Beyond this, custom benchmarking code is fun but open ended, and this section must stop here for space. As next steps, you might modify the timing script to measure the speed of *set* and *dictionary* comprehensions and their for equivalents (open questions we'll return to ahead); explore Python's `profile` and `cProfile` modules (tools that create full-program profiles instead of focused benchmarks); or explore Python's `timeit` module, which works much like some of this chapter's homegrown code, and offers extra options—as you'll learn in the next section.



Decorator timers preview: Notice how we must pass functions into the timers manually here. In [Chapter 39](#), we'll code *decorator*-based timer alternatives with which timed functions are called normally, but require extra `@` preamble syntax where defined. Decorators may be more useful to instrument functions with timing logic when they are already being used within a larger system, and don't as easily support the specific test-call patterns assumed here—when decorated, *every* call to the function runs the timing logic, which is either a plus or minus depending on your goals.

Benchmarking with Python's `timeit`

The preceding section used custom timing functions to compare code speed. As teased there, the Python standard library also ships with a module named `timeit` that can be used in similar ways, but offers added flexibility, with support for timing code strings in addition to functions, as well as a command-line mode.

As usual in Python, it's important to understand fundamental principles like those illustrated in the prior section. Python's “batteries included” approach means you'll usually find precoded options for most goals, though you still need to know the ideas underlying them to choose and use them properly. Indeed, the `timeit` module is a prime example of this—it's had a history of being misused by newcomers who didn't yet understand the concepts it embodies. Now that we've learned the basics, though, let's move ahead to a tool that can automate much of our work.

Basic `timeit` Usage

Let's start with this module's fundamentals before leveraging them in larger scripts. With `timeit`, tests are specified by either *callable objects* or *statement strings*; the latter can hold multiple statements if they use `;` separators or `\n` characters for line breaks, and spaces or tabs to indent statements in nested blocks (e.g., `\n\t`). Tests may also give setup actions, and can be launched from both *command lines* and *API calls*, and from both scripts and the REPL.

API-calls mode

For example, the `timeit` module's `repeat` call returns a list giving the total time taken to run a test a number of times, for each of `repeat` runs—the `min` of this list yields the best time among the runs, and

helps filter out system load fluctuations that can otherwise skew timing results artificially high (like our earlier `bestoftotal`). Code to be timed is passed to the `stmt` keyword (or first positional) argument, and may be a string or no-argument function.

The following shows this call in action in REPLs on macOS, timing a list comprehension on both CPython 3.12 and the optimized PyPy implementation of Python described in [Chapter 2](#) (as earlier, its tested 7.3 release implements the Python 3.10 language). The results here give the best total time in seconds among 5 runs that each execute the code string 1,000 times; the code string itself constructs a 1,000-item list of integers each time through (`timeit` also has reasonable defaults for repeat counts, but being explicit ensures comparability):

```
$ python3
>>> import timeit
>>> min(timeit.repeat(stmt='[x ** 2 for x in range(1000)]', number=1000, repeat=5))
0.05187674192711711

$ pypy3
>>>> import timeit
>>>> min(timeit.repeat(stmt='[x ** 2 for x in range(1000)]', number=1000, repeat=5))
0.00252467580139637
```

You'll notice that PyPy checks in at *20X* faster than CPython 3.12. This is a small artificial benchmark, of course, but stunning nonetheless, and reflects a relative speed ranking that is generally supported by other tests run in this book (though CPython will still beat PyPy on some types of code ahead, and again, may improve with a future JIT). To be fair, this particular test measures the speed of both a list comprehension and integer math, but integer math is ubiquitous in Python code, and PyPy's win is larger on noninteger tests (try squaring a float to see for yourself).

These results also differ from the preceding section's relative version speeds, where PyPy was some *10X* quicker for list comprehensions. Apart from the different type of code being timed here, the different coding structure inside `timeit` may have an effect too—for code strings like those tested here, `timeit` builds, compiles, and executes a function `def` statement string that embeds the test string, thereby avoiding a function call per inner loop. This is irrelevant from a relative-speed perspective, though: times from the same given tool are comparable.

You'll also notice that code to be timed is a *string* here. As you'll see ahead, this requires manually coded line breaks and indentation in some usage modes, and all code to be timed this way must conform to Python's rules for string literals, quotes, and escapes. For instance, a code string cannot embed the same quotes used to enclose it without also escaping them. You can avoid this potential downside by timing callables, though they're limited to zero arguments.

Command-line mode

The `timeit` module also can be run as a script from a command line—either by explicit pathname, or, more portably, automatically located on the module search path with Python's `-m` switch (we used this earlier to launch `PyDoc` in [Chapter 15](#) and `IDLE` in [Chapter 3](#)). In this mode, you'll need to *quote or escape* Python code in the command line per your console shell's rules; double quotes are generally portable, but this also qualifies as a potential downside.

Also in this mode, `timeit` reports the average time for a *single* `-n` loop, in either “usec” microseconds (millionths), “msec” milliseconds (thousandths), “sec” seconds, or “nsec” nanoseconds; to compare results here to the total time values reported by API calls, multiply by the number of loops run—51 usec here * 1,000 loops is 51k usec, 51 msec, and 0.051 seconds in total time. For CPython 3.12 on macOS:

```
$ python3 -m timeit -n 1000 "[x ** 2 for x in range(1000)]"
1000 loops, best of 5: 51.9 usec per loop

$ python3 -m timeit -n 1000 -r 50 "[x ** 2 for x in range(1000)]"
1000 loops, best of 50: 51.8 usec per loop
```

As another example, we can use command lines to verify that choice of timer call doesn't impact speed comparisons run in this chapter so far. `timeit` uses `time.perf_counter` by default but its `-p` flag instructs it to instead use `time.process_time`, both discussed earlier (spoiler: as tested, there's no discernible difference):

```
$ python3 -m timeit -p -n 1000 -r 50 "[x ** 2 for x in range(1000)]"
1000 loops, best of 50: 51.8 usec per loop
```

We can also use `timeit` to see how PyPy stacks up again—but we're in for a bit of a surprise:

```
$ pypy3 -m timeit -n 1000 -r 50 "[x ** 2 for x in range(1000)]"
WARNING: timeit is a very unreliable tool. use pyperf or something
else for real measurements
pypy3 -m pip install pyperf
pypy3 -m pyperf timeit -n '1000' -r '50' '[x ** 2 for x in range(1000)]'
-----
1000 loops, average of 50: 1.54 +- 0.568 usec per loop (using standard deviation)
```

As you can see, PyPy has customized command-line mode in its version of `timeit` to both emit a subjective redirect to an alternative timer, as well as modify the result display to show averages instead of minimums. These are both opinionated mods made since this book's prior edition, though perhaps understandable for a tool so focused on (and sensitive to) benchmark results. See the web for more on the suggested *pyperf*; it's not part of Python's standard library and requires a separate install, but may offer more advanced timing options that may or may not apply to your goals.

Handling multiline statements

Happily, `timeit`'s API mode is still opinion-free today. To time larger blocks of code in *API-call* mode, either load code from a file, use a triple-quoted block string, or use newlines and tabs or spaces to satisfy Python's syntax. Because you pass Python string objects to a Python function in this mode, there are no shell considerations, though be careful to handle nested quotes with escapes or mixed quotes if needed. The following, for instance, times [Chapter 13](#) loop alternatives in CPython 3.12; you can use the same pattern to time the file-line-reader alternatives in [Chapter 14](#):

```
$ python3
>>> import timeit
>>> min(timeit.repeat(number=100_000, repeat=5,
    stmt='L = [1, 2, 3, 4, 5]\nfor i in range(len(L)): L[i] += 1'))
0.028153221122920513

>>> min(timeit.repeat(number=100_000, repeat=5,
    stmt='L = [1, 2, 3, 4, 5]\ni=0\nwhile i < len(L):\n\tL[i] += 1\n\ti += 1'))
0.03337613819167018

>>> min(timeit.repeat(number=100_000, repeat=5,
    stmt='L = [1, 2, 3, 4, 5]\nM = [x + 1 for x in L]'))
0.021507765166461468
```

To run multiline statements like these in *command-line* mode, appease your shell by passing each statement line as a separate argument, with whitespace for indentation—`timeit` concatenates all the lines together with a newline character between them, and later re-indents for its own statement-nesting purposes. Leading spaces may work better for indentation than tabs in this mode due to shell variability,

and be sure to quote the code arguments if required by your shell (the first of the following was line-split here to fit, but must be input on a single line in some shells):

```
$ python3 -m timeit -n 100000 -r 5 "L = [1,2,3,4,5]" "i=0" "while i < len(L):"
"    L[i] += 1" "    i += 1"
100000 loops, best of 5: 332 nsec per loop

$ python3 -m timeit -n 100000 -r 5 "L = [1,2,3,4,5]" "M = [x + 1 for x in L]"
100000 loops, best of 5: 218 nsec per loop
```

Other timeit usage modes

The `timeit` module also allows you to provide *setup* code that is run in the main statement's scope, but whose time is not charged to the main statement's total—useful for initialization code you wish to exclude from total time, such as imports of required modules and builds of test data. Because they're run in the same scope, any names created by setup code are available to the main test statement; names defined in the interactive shell generally are not.

To specify setup code, use a `-s` in command-line mode (or many of these for multiline setups) and a `setup` argument string in API-call mode. This can focus tests more sharply, as in the following, whose second command splits list initialization off to a setup statement to time just iteration in command-line mode. As a general rule of thumb, though, the more code you include in a test statement, the more applicable its results will generally be to realistic code:

```
$ python3 -m timeit -n 100000 -r 5 "L = [1,2,3,4,5]" "M = [x + 1 for x in L]"
100000 loops, best of 5: 211 nsec per loop

$ python3 -m timeit -n 100000 -r 5 -s "L = [1,2,3,4,5]" "M = [x + 1 for x in L]"
100000 loops, best of 5: 175 nsec per loop
```

Timing sort speed

To demo setup code in API-call mode, the interaction that follows times a sort-based option in [Chapter 18](#)'s minimum-value example. To avoid page flipping, here's the function it times, in the module `mins.py` ([Example 18-2](#)):

```
def min4(*args):
    return sorted(args)[0]
```

And here are the results—proving indirectly that sequences *sort much faster* when they are ordered than they do with randomly ordered contents (to see for yourself, run this in [Chapter 18](#)'s code folder):

```
>>> from timeit import repeat                                     # Standard-library module

>>> min(repeat(number=1000, repeat=5,                             # Sort an ordered list in min4
          setup='from mins import min4\n'                        # Adjacent strings concatenated
          'vals=list(range(1000))',                               # Code within () spans lines
          stmt= 'min4(*vals)'))                                    # First import prints output
0.011066518723964691

>>> min(repeat(number=1000, repeat=5,                             # Sort a randomly ordered list
          setup='from mins import min4\n'
          'import random\n'
          'vals=[random.random() for i in range(1000)]',
          stmt= 'min4(*vals)'))
0.068130933213979
```

See Python's manuals for more on the `random` module used here and in earlier chapters, as well as more on `timeit`. Not shown here, `timeit` also lets you ask for just total time, time callable objects instead of strings, use a class-based API, and leverage additional command-line switches and API-call arguments we don't have space to cover. Instead, the next section codes a `timeit` utility that goes beyond manual command lines and REPL calls.

Automating timeit Benchmarking

Rather than going into more `timeit` details, let's study a program that deploys it to time both coding alternatives and Python versions. This will let us easily define a set of tests to time in a separate file, and will allow us collect data from multiple Pythons, both individually and grouped (though grouped mode comes with limits today, as you'll see).

Benchmark module

To get started, the module file in [Example 21-8](#), `pybench.py`, is set up to time a sequence of statements coded in scripts that import and use it, with either the Python running its code or all Python versions named in a list. It uses some application-level tools described ahead. Because it mostly applies ideas we've already explored and is amply documented, though, it's listed as mostly self-study material, and an exercise in reading Python code.

Example 21-8. pybench.py

```
r"""
Time the speed of one or more Pythons on multiple code-string
benchmarks with timeit. This is a function, to allow timed tests
to vary. It times all code strings in a passed list, in either:

1) The Python running this script, by timeit API calls
2) Multiple Pythons whose paths are passed in a list, by reading
   the output of timeit command lines run by os.popen that use
   Python's -m switch to find timeit on the module search path

In command-line mode (2) only, this replaces all " in timed code
with ', to avoid clashes with argument quoting; splits multiline
statements into one quoted argument per line so all will be run;
and replaces all \t in indentation with 4 spaces for uniformity.

Caveats:
- Command-line mode (only) uses naive quoting and MAY FAIL if code
  embeds and requires double quotes; quoted code is incompatible
  with the host shell; or command length exceeds shell limits.
- PyPy is largely unusable in command-line mode (2) today, as its
  modified timeit output in this mode is jarring in the report.
- This does not (yet?) support a setup statement in any mode: the
  time of all code in the test stmt is charged to its total time.

As fallbacks on fails, use either this module's API-call mode to
test one Python at a time, or the homegrown timer.py module.
"""

import sys, os, time, timeit
defnum, defrep= 1000, 5    # May vary per stmt

def show_context():
    """
```

```

Show run's context using an arguably gratuitous f-string
that fails on 3.10 PyPy without "... " for nested ' quotes.
"""
print(f"Python {'.'.join(str(x) for x in sys.version_info[:3])}"
      f' on {sys.platform}'
      f" at {time.strftime('%b-%d-%Y, %H:%M:%S')}")

def runner(stmts, pythons=None, tracecmd=False):
    """
    Main logic: run tests per input lists which determine usage modes.
    stmts: [(number?, repeat?, stmt-string)]
    pythons: None=host python only, or [python-executable-paths]
    """
    show_context()
    for (number, repeat, stmt) in stmts:
        number = number or defnum
        repeat = repeat or defrep    # 0=default

        if not pythons:
            # Run stmt on this python: API call
            # No need to split lines or quote here
            best = min(timeit.repeat(stmt=stmt, number=number, repeat=repeat))
            print(f'{best:.4f} {stmt[:70]!r}')

        else:
            # Run stmt on all pythons: command line
            # Split lines into quoted arguments
            print('-' * 80)
            print(repr(stmt))                                # show quotes
            for python in pythons:
                stmt = stmt.replace('"', "'")                # all " => '
                stmt = stmt.replace('\t', ' ' * 4)           # tab => ____
                lines = stmt.split('\n')                     # line => arg
                args = ' '.join(f'"{line}"' for line in lines) # arg => "arg"

                oscmd = f'{python} -m timeit -n {number} -r {repeat} {args}'
                print(oscmd if tracecmd else python)
                print('\t' + os.popen(oscmd).read().rstrip())

```

Benchmark script

This preceding file is really only half the picture. Testing scripts use this module's function, passing in concrete though variable lists of statements and Pythons to be tested, as appropriate for the usage mode desired. For example, the script in [Example 21-9](#), *pybench_tests.py*, tests a handful of statements and Pythons by importing and using [Example 21-8](#), and allows command-line arguments to determine part of its operation: `-a` tests all listed Pythons instead of just one, and an added `-t` traces constructed command lines in full so you can see how quotes, multiline statements, and tabs are handled per `timeit`'s command-line formats shown earlier (see both files' docstrings for more details).

Example 21-9. *pybench_tests.py*

```

"""
Run pybench.py to time one or more Pythons on multiple code strings.
Use command-line arguments (which appear in sys.argv) to select modes:

<python> pybench_tests.py
    times just the hosting Python on all code listed in stmts below
python3 pybench_tests.py -a

```

```

    times all stmts in all pythons whose paths are listed below
python3 pybench_tests.py -a -t
    same as -a, but also traces command lines in full

```

Edit stmts below to change tested code, and edit pythons below to give paths of Python executables to be tested in -a mode. To find a Python's path, start its REPL, run "import sys", and inspect "sys.executable".

```

import pybench, sys

pythons = [
    '/Library/Frameworks/Python.framework/Versions/3.12/bin/python3',
    '/Users/me/Downloads/pypy3.10-v7.3.16-macos_x86_64/bin/pypy3',
]

stmts = [
    # Iterations
    (0, 0, '[x ** 2 for x in range(1000)]'),          # (num,rpt,stmt)
    (0, 0, 'res=[]\nfor x in range(1000): res.append(x ** 2)'), # |n=multistmt
    (0, 0, 'list(map(lambda x: x ** 2, range(1000)))'), # |n|t=indent
    (0, 0, 'list(x ** 2 for x in range(1000))'),
    # String ops
    (0, 0, "s = 'hack' * 2500\nx = [s[i] for i in range(10_000)]"),
    (0, 0, "s = '?'\nfor i in range(10_000): s += '?'"),      # A PyPy loss!
]

tracecmd = '-t' in sys.argv          # -t: trace command lines?
pythons = pythons if '-a' in sys.argv else None # -a: all in list, else one?
pybench.runner(stmts, pythons, tracecmd)      # Time pythons on all stmts

```

Timing individual Pythons

The following is the preceding script's output when run to test a *specific Python*—the one running the script. This mode uses direct `timeit` API calls, not command lines, with total time listed in the left column, the statement tested quoted on the right, and a context line at the top courtesy of Python's `sys` and `time` modules (see its manuals for more info). These two runs again use CPython 3.12 and PyPy 7.3 (i.e., 3.10), respectively, on the same host:

```

$ python3 pybench_tests.py
Python 3.12.2 on darwin at Jun-27-2024, 15:21:02
0.0533 '[x ** 2 for x in range(1000)]'
0.0605 'res=[]\nfor x in range(1000): res.append(x ** 2)'
0.0804 'list(map(lambda x: x ** 2, range(1000)))'
0.0759 'list(x ** 2 for x in range(1000))'
0.4042 "s = 'hack' * 2500\nx = [s[i] for i in range(10_000)]"
0.8061 "s = '?'\nfor i in range(10_000): s += '"

$ pypy3 pybench_tests.py
Python 3.10.14 on darwin at Jun-27-2024, 15:22:14
0.0020 '[x ** 2 for x in range(1000)]'
0.0040 'res=[]\nfor x in range(1000): res.append(x ** 2)'
0.0030 'list(map(lambda x: x ** 2, range(1000)))'
0.0077 'list(x ** 2 for x in range(1000))'
0.0529 "s = 'hack' * 2500\nx = [s[i] for i in range(10_000)]"
1.2942 "s = '?'\nfor i in range(10_000): s += '"

```

Drawing conclusions from this is left as a suggested exercise, but notice that PyPy actually *loses* to CPython on the very last test run. Again, performance can vary per code, and absolutes in benchmarking are perilous at best.

Timing multiple Pythons

As noted, this script can also test multiple Pythons for each statement string, by adding a `-a` to the command line. In this mode the script itself is run by CPython 3.12, which launches shell command lines that start other Pythons to run the `timeit` module on the statement strings. To make this work, this mode must split, format, and quote multiline statements for use in command lines according to both `timeit` expectations and shell requirements.

This mode also relies on the `-m` Python command-line flag to locate `timeit` on the module search path and run it as a script, as well as the `os.popen` and `sys.argv` standard-library tools to run a shell command and inspect command-line arguments, respectively. Add a `-t` to trace commands run, and see Python manuals and other resources for more on these tools; `os.popen` is also mentioned briefly in Chapters 3, 9, and 13. Here is this mode's result:

```
$ python3 pybench_tests.py -a
Python 3.12.2 on darwin at Jun-27-2024, 16:12:39
-----
'[x ** 2 for x in range(1000)]'
/Library/Frameworks/Python.framework/Versions/3.12/bin/python3
1000 loops, best of 5: 52.7 usec per loop
/Users/me/Downloads/pypy3.10-v7.3.16-macos_x86_64/bin/pypy3
WARNING: timeit is a very unreliable tool. use pyperf or something
else for real measurements
pypy3 -m pip install pyperf
pypy3 -m pyperf timeit -n '1000' -r '5' '[x ** 2 for x in range(1000)]'
-----
1000 loops, average of 5: 2.66 +- 1.55 usec per loop (using standard deviation)
...more output clipped...
```

Regrettably, this all-Pythons mode is nearly unusable today: the opinionated inserts and mods made to `timeit` by PyPy render its command-line output very different from the norm, and very difficult to read or list here. Moreover, there's no way to disable these (short of dropping CPython's `timeit.py` into PyPy's library folder, which seems too much for this demo to ask). Hence, while this script's `-a` mode still works in this edition, it may be best used to time a set of Pythons that do not subjectively mod the behavior of a widely used standard-library module like `timeit`.

Timing set and dictionary iterations

The good news is that *single-Python* mode still allows us to easily script a set of benchmarks—even on PyPy. The script in [Example 21-10](#), for instance, uses the driver module in [Example 21-8](#) to see how sets and dictionaries fare.

Example 21-10. *pybench_tests2.py*

```
import pybench

stmts = [
    # Sets
    (0, 0, '{x ** 2 for x in range(1000)}'),
    (0, 0, 'set(x ** 2 for x in range(1000))'),
    (0, 0, 's=set()\nfor x in range(1000): s.add(x ** 2)'),
    # Dicts
    (0, 0, '{x: x ** 2 for x in range(1000)}'),
    (0, 0, 'dict((x, x ** 2) for x in range(1000))'),
    (0, 0, 'd={} \nfor x in range(1000): d[x] = x ** 2'),
]

pybench.runner(stmts, None, False)    # No -a mode in this script
```

As suggested in the preceding chapter, passing a generator to a type name is indeed substantially slower than other construction schemes—as this script’s output on both CPython 3.12 and PyPy 7.3 (3.10) finally proves:

```
$ python3 pybench_tests2.py
Python 3.12.2 on darwin at Jun-27-2024, 16:20:19
0.0746 '{x ** 2 for x in range(1000)}'
0.0947 'set(x ** 2 for x in range(1000))'
0.0834 's=set()\nfor x in range(1000): s.add(x ** 2)'
0.0745 '{x: x ** 2 for x in range(1000)}'
0.1174 'dict((x, x ** 2) for x in range(1000))'
0.0754 'd={} \nfor x in range(1000): d[x] = x ** 2'

$ pypy3 pybench_tests2.py
Python 3.10.14 on darwin at Jun-27-2024, 16:22:18
0.0191 '{x ** 2 for x in range(1000)}'
0.0222 'set(x ** 2 for x in range(1000))'
0.0186 's=set()\nfor x in range(1000): s.add(x ** 2)'
0.0188 '{x: x ** 2 for x in range(1000)}'
0.0398 'dict((x, x ** 2) for x in range(1000))'
0.0185 'd={} \nfor x in range(1000): d[x] = x ** 2'
```

Conclusion: Comparing tools

If you have time, check out *pybench_tests3.py* in the book example’s package (omitted here for space) for another test that verifies that this section’s *timeit* tools turn in results similar to the earlier *timer.py* homegrown tools. Its findings generally jive with those in “[Iteration Results](#)” on [page 535](#), though they arrive at them by very different means:

```
$ python3 pybench_tests3.py
Python 3.12.2 on darwin at Jun-27-2024, 16:26:44
0.2766 "res=[]\nfor x in 'hack' * 2500: res.append(ord(x))"
0.2173 "[ord(x) for x in 'hack' * 2500]"
0.1430 "list(map(ord, 'hack' * 2500))"
0.4172 "list(ord(x) for x in 'hack' * 2500)"
```

For more fidelity, study this chapter’s code and run more tests on your own. Benchmarks can be great sport, but we’ll have to leave further excursions as suggested exercises. Here, it’s time to wrap up this chapter and part, so we can move on to learn more about the modules we’ve been using informally since [Chapter 16](#).



Cross-platform results: As a bonus, folder `_benchmark-platform-results` in the book's examples package collects CPython's results for this chapter's *timer* and *pybench* tests on multiple platforms—macOS, Windows, Android, and Linux. Due to Python and host differences, not all its results are directly comparable, but they are relatively similar. Surprisingly, an Android foldable phone beats all the PCs, though these tests are CPU bound, and operations like file access may fare differently. Caveat: these results *will* change; retest with your Pythons and devices for current data.

Function Gotchas

Now that we've reached the end of the function story, let's review some common pitfalls. Functions have some jagged edges that you might not expect. They're all relatively obscure, and a few have started to fall away from the language completely in recent releases, but most have been known to trip up new users.

Local Names Are Detected Staticly

As you've learned, Python classifies names assigned in a function as *locals* by default; they live in the function's scope and exist only while the function is running. What you may not realize is that Python detects locals statically, when it compiles the `def`'s code, rather than by noticing assignments as they happen at runtime. This leads to one of the most common oddities posted on the Python newsgroup by beginners.

Normally, a name that isn't assigned in a function is looked up in the enclosing module:

```
>>> X = 99

>>> def selector():           # X used but not assigned
    print(X)                  # X found in global scope

>>> selector()
99
```

Here, the `X` in the function resolves to the `X` in the module. But watch what happens if you add an assignment to `X` after the reference:

```
>>> def selector():
    print(X)                  # Does not yet exist!
    X = 88                    # X classified as a local name (everywhere)
                                # Can also happen for "import X", "def X"...

>>> selector()
UnboundLocalError: cannot access local variable 'X' where it is not associated with a value
```

You get the name-usage error shown here, but the reason is subtle. Python reads and compiles this code when it's typed interactively or imported from a module. While compiling, Python sees the assignment to `X` and decides that `X` will be a local name everywhere in the function. But when the function is actually run, because the assignment hasn't yet happened when the `print` executes, Python says you're using an undefined name. According to its name rules, it should say this; the local `X` is used before being assigned. In fact, any assignment in a function body makes a name local. Imports, `=`, nested `defs`, nested classes, and so on are all susceptible to this behavior.

The problem occurs because assigned names are treated as locals everywhere in a function, not just after the statements where they're assigned. Really, the previous example is ambiguous: was the

intention to print the global `X` and create a local `X`, or is this a real programming error? Because Python treats `X` as a local everywhere, it's seen as an error; if you mean to print the global `X`, you need to declare it in a `global` statement:

```
>>> def selector():
    global X                # Force X to be global (everywhere in function)
    print(X)
    X = 88

>>> selector()
99
```

Remember, though, that this means the assignment also changes the global `X`, not a local `X`. Within a function, you can't use both local and global versions of the same simple name. If you really meant to print the global and then set a local of the same name, you'd need to import the enclosing module and use module attribute notation to get to the global version:

```
>>> X = 99
>>> def selector():
    import __main__         # Import enclosing module
    print(__main__.X)       # Qualify to get to global version of name
    X = 88                  # Unqualified X classified as local
    print(X)                # Prints local version of name

>>> selector()
99
88
```

Qualification (the `.X` part) fetches a value from a namespace object. The interactive namespace is a module called `__main__`, so `__main__.X` reaches the global version of `X`. If that isn't clear, review [Chapter 17](#).

In recent versions, Python has improved on this story somewhat by issuing for this case the more specific “unbound local” error message shown in the example listing (it used to simply raise a generic name error); this gotcha is still present in general, though.

Defaults and Mutable Objects

As noted briefly in Chapters [17](#) and [18](#), mutable values for default arguments can retain state between calls, though this is often unexpected. In general, default argument values are evaluated and saved *once* when a `def` statement is run, not each time the resulting function is later called. Internally, Python saves one object per default argument, attached to the function itself.

That's usually what you want—because defaults are evaluated at `def` time, they let you save values from the enclosing scope, if needed (functions defined within loops by factories may even depend on this behavior—see ahead). But because a default retains an object between calls, you have to be careful about changing mutable defaults. For instance, the following function uses an empty list as a default value, and then changes it in place each time the function is called:

```
>>> def saver(x=[]):
    # Saves away a list object
    x.append(1)      # Changes same object each time!
    print(x)

>>> saver([2])      # Default not used
[2, 1]
>>> saver()          # Default used
[1]
```



```
>>> saver()                                # Grows on each call!
[1, 1]
>>> saver()
[1, 1, 1]
```

Some see this behavior as a feature—because mutable default arguments retain their state between function calls, they can serve some of the same roles as “static” local function variables in the C language. In a sense, they work much like global variables, but their names are local to the functions and so will not clash with names elsewhere in a program.

To other observers, though, this seems like a gotcha—especially the first time they run into it. There are better ways to retain state between calls in Python (e.g., using the nested scope closures and function attributes we met in this part, and the classes we will study in [Part VI](#)).

Moreover, mutable defaults can be overridden by real values, and are tricky to remember (and to understand at all). They depend upon the timing of default object construction. In the prior example, there is just one list object for the default value—the one created when the `def` is executed. You don’t get a new list every time the function is called, so the list grows with each new `append`; it is not reset to empty each time.

If that’s not the behavior you want, simply make a copy of the default at the start of the function body, or move the default value expression into the function body. As long as the value resides in code that’s actually executed each time the function is *called*, you’ll get a new object each time through:

```
>>> def saver(x=None):
    if x is None:                # No argument passed?
        x = []                  # Run code to make a new list each time
    x.append(1)                  # Changes new list object
    print(x)

>>> saver([2])
[2, 1]
>>> saver()                    # Doesn't grow here
[1]
>>> saver()
[1]
```

By the way, the `if` statement in this example could *almost* be replaced by the assignment `x = x or []`, which takes advantage of the fact that Python’s `or` returns one of its operand objects: if no argument was passed, `x` would default to `None`, so the `or` would return the new empty list on the right.

However, this isn’t exactly the same. If an empty list were passed in, the `or` expression would cause the function to extend and return a newly created list, rather than extending and returning the passed-in list like the `if` version. (The expression becomes `[] or []`, which evaluates to the new empty list on the right; see [Chapter 12](#)’s “Truth Values Revisited” if you don’t recall why.) Real program requirements may call for either behavior.

Today, another way to achieve the value retention effect of mutable defaults in a possibly less confusing way is to use the *function attributes* we discussed in [Chapter 19](#):

```
>>> def saver():
    saver.x.append(1)
    print(saver.x)

>>> saver.x = []
>>> saver()
[1]
>>> saver()
```

```
[1, 1]
>>> saver()
[1, 1, 1]
```

The function name is global to the function itself, but it need not be declared because it isn't changed directly within the function. This isn't used in exactly the same way, but when coded like this, the attachment of an object to the function is much more explicit (and arguably less magical).

Functions Without returns

In Python functions, `return` (and `yield`) statements are optional. When a function doesn't return a value explicitly, the function exits when control falls off the end of the function body. Technically, all functions return a value; if you don't provide a `return` statement, your function returns the `None` object automatically:

```
>>> def proc(x):
    print(x)                                # No return is a None return

>>> x = proc('testing 123...')
testing 123...
>>> print(x)
None
```

Functions such as this without a `return` are Python's equivalent of what are called “procedures” in some languages. They're usually invoked as statements, and the `None` results are ignored, as they do their business without computing a useful result.

This is worth remembering, because Python won't tell you if you try to use the result of a function that doesn't return one. As we noted in [Chapter 11](#), for instance, assigning the result of a list `append` method won't raise an error, but you'll get back `None`, not the modified list:

```
>>> list = [1, 2, 3]
>>> list = list.append(4)                   # append is a "procedure"
>>> print(list)                             # append changes list in place
None
```

“[Common Coding Gotchas](#)” on [page 372](#) discusses this more broadly. In general, any functions that do their business as a side effect are usually designed to be run as statements, not expressions.

Miscellaneous Function Gotchas

Here are two additional function-related gotchas—mostly reviews, but common enough to reiterate.

Enclosing scopes and loop variables

We met this gotcha in [Chapter 17](#)'s discussion of enclosing function scopes, but as a reminder: when coding factory functions (a.k.a. closures), be careful about relying on enclosing function scope lookup for variables that are changed by enclosing loops. When a generated function is later called, all such references will remember the value of the *last* loop iteration in the enclosing function's scope. In this case, you must use *defaults* to save loop variable values instead of relying on automatic lookup in enclosing scopes. See “[Loops Require Defaults, Not Scopes](#)” on [page 418](#) in [Chapter 17](#) for more details on this topic.

Hiding built-ins by assignment

Also in [Chapter 17](#), we saw how it's possible to reassign built-in names in a closer local or global scope; the reassignment effectively hides (“shadows”) and replaces that built-in's name for the remainder of the scope where the assignment occurs. This means you won't be able to use the original built-in value for the name. As long as you don't need the built-in value of the name you're assigning, this isn't an issue—many names are built in, and they may be freely reused. However, if you reassign a built-in name your code relies on, you may have problems. So don't do that, unless you really mean to. The good news is that the built-ins you commonly use will soon become second nature, and Python's error trapping will alert you early in testing if your built-in name is not what you think it is.

Chapter Summary

This chapter rounded out our look at functions and built-in iteration tools with a larger case study that measured the performance of iteration alternatives and other tools we've met along the way, as well as one alternative Python implementation. We used both custom timer code as well as Python's `timeit` with a helper to time code in a variety of modes. We also reviewed common function-related mistakes to help you avoid pitfalls.

This concludes the functions part of this book. The next part expands on what we already know about *modules*—files of tools that form the topmost organizational unit in Python programs, and the structure in which functions reside. After that, we will explore *classes*, which are largely packages of functions with special first arguments. As you'll see, classes can implement objects that tap into the iteration protocol, just like the generators and iterables we used here. In fact, everything we have learned in this part of the book will apply when functions take the guise of class methods.

Before moving on to modules, though, be sure to work through this chapter's quiz, as well as the exercises for this part of the book, to practice what you've learned about functions here.

Test Your Knowledge: Quiz

1. What conclusions can you draw from this chapter about the relative speed of Python iteration tools?
2. What conclusions can you draw from this chapter about the relative speed of the Pythons timed?

Test Your Knowledge: Answers

1. In CPython today, list comprehensions are often the quickest of the bunch; `map` beats list comprehensions in Python when all tools must call built-in functions; for loops tend to be slower than comprehensions; and generator functions and expressions are slower than all other options. Under PyPy, some of these findings differ; `map` often turns in a different relative performance, for example, and list comprehensions seem always quickest, perhaps due to function-level optimizations.

At least that's the case today on the Python versions tested, on the test machine used, and for the type of code timed—these results may vary if any of these three variables differ. Use the home-grown `timer` or standard library `timeit` to test your use cases for more relevant results. Also keep in mind that iteration is just one component of a program's time: more code gives a more complete picture.

2. In general, PyPy 7.3 (implementing Python 3.10) is substantially faster than CPython 3.12. In some cases timed, PyPy was 5X–20X faster than CPython, though in isolated cases (e.g., some string operations), PyPy was slower than CPython, and PyPy’s use of a JIT can impact benchmark results.

At least that’s the case today on the Python versions tested, on the test machine used, and for the type of code timed—these results may vary if any of these three variables differ. Use the home-grown `timer` or standard library `timeit` to test your use cases for more relevant results. This is especially true when timing Python implementations, which may be arbitrarily optimized in each new release—in fact, CPython may soon adopt a JIT like PyPy, which could invalidate results here. We also didn’t test any of the many other Python implementations; see [Chapter 2](#) for other options to time on your own.

Test Your Knowledge: Part IV Exercises

In these exercises, you’re going to start coding more sophisticated programs. Be sure to check the solutions in “[Part IV, Functions and Generators](#)” on [page 1167](#) in [Appendix B](#), and be sure to start writing your code in module files. You won’t want to retype these exercises in a REPL if you make a mistake.

1. *The basics*: At the Python interactive prompt, write a function named `echo` that prints its single argument to the screen and call it interactively, passing a variety of object types: string, integer, list, dictionary. Then, try calling it without passing any argument. What happens? What happens when you pass two arguments?
2. *Arguments*: Write a function called `adder` in a Python module file named `adder1.py`. The function should accept two arguments and return the sum (or concatenation) of the two. Then, add code at the bottom of the file to call the `adder` function with a variety of object types (two strings, two lists, two floating points), and run this file as a script from the system command line. Do you have to print the call statement results to see results on your screen?
3. *Arbitrary arguments*: Copy the file you wrote in the last exercise to `adder2.py`, generalize its `adder` function to compute the sum of an arbitrary number of arguments, and change the calls to pass more or fewer than two arguments. What type is the returned sum? (Hints: a slice such as `S[:0]` returns an empty sequence of the same type as `S`, and the `type` built-in function can test types; but see the manually coded `min` examples in [Chapter 18](#) for a simpler approach.) What happens if you pass in arguments of different types? What about passing in dictionaries?
4. *Keywords*: In an `adder3.py`, change the `adder` function from exercise 2 to accept and sum/concatenate three arguments: `def adder(red, green, blue)`. Now, provide *default* values for each argument, and experiment with calling the function interactively or code tests in the file. Try passing one, two, three, and four arguments. Then, try passing *keyword* arguments. Does the call `adder(blue=1, red=2)` work? Why? Finally, copy and generalize the new `adder` to accept and sum/concatenate an *arbitrary* number of keyword arguments in an `adder4.py`. This is similar to what you did in exercise 3, but you’ll need to iterate over a dictionary, not a tuple. (Hint: the `dict.keys` method returns an iterable you can step through with a `for` or `while`, but be sure to wrap it in a `list` call to index it; `dict.values` may help here too.)
5. *Dictionary tools*: Write a function called `copyDict(dict)` that copies its dictionary argument. It should return a new dictionary containing all the items in its argument. Use the dictionary `keys` method to iterate (or step over a dictionary’s keys without calling `keys`). Copying sequences is easy (`X[:]` makes a top-level copy); does this work for dictionaries, too? As explained in this exercise’s solution, because dictionaries come with similar tools, this and the next exercise are just coding exercises but still serve as representative functions.

6. *Dictionary tools*: Write a function called `addDict(dict1, dict2)` that computes the union (i.e., merge) of two dictionaries. It should return a new dictionary containing all the items in both its arguments (which are assumed to be dictionaries). If the same key appears in both arguments, feel free to pick a value from either. Test your function by writing it in a file and running the file as a script. What happens if you pass lists instead of dictionaries? How could you generalize your function to handle this case, too? (Hint: see the type built-in function used earlier.) Does the order of the arguments passed in matter? Dictionary merge is also a built-in today (actually, several), but you're trying to stretch yourself a bit by coding it manually.
7. *More argument-matching examples*: First, define the following six functions (either interactively or in a module file that can be imported):

```
def f1(a, b): print(a, b)           # Normal args

def f2(a, *b): print(a, b)         # Positional collectors

def f3(a, **b): print(a, b)        # Keyword collectors

def f4(a, *b, **c): print(a, b, c) # Mixed modes

def f5(a, b=2, c=3): print(a, b, c) # Defaults

def f6(a, b=2, *c): print(a, b, c) # Defaults and positional collectors
```

Now, test the following calls interactively, and try to explain each result; in some cases, you'll probably need to fall back on the matching rules covered in [Chapter 18](#). Do you think mixing matching modes is a good idea in general? Can you think of cases where it would be useful?

```
>>> f1(1, 2)
>>> f1(b=2, a=1)

>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, **dict(x=2, y=3))

>>> f5(1)
>>> f5(1, 4)
>>> f5(1, c=4)

>>> f6(1)
>>> f6(1, *[3, 4])
```

8. *Primes revisited*: Recall the following code snippet from [Chapter 13](#), which simplistically determines whether a positive integer is prime:

```
x = num // 2                      # For some num > 1, start at half
while x > 1:
    if num % x == 0:               # Remainder 0? Factor found
        print(num, 'has factor', x)
        break                     # Exit now and skip else
    x -= 1
else:                              # Normal exit, when x reaches 1
    print(num, 'is prime')
```

Package this code as a reusable function in a module file (`num` should be a passed-in argument), and add some calls to the function at the bottom of your file to test. While you're at it, experiment with replacing the first line's `//` operator with `/` to see how *true* division breaks this code (refer back to [Chapter 5](#) if you need a reminder). What can you do about negatives, and the values 0 and 1? How about speeding this up? Your outputs should look something like this:

```
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
```

9. *Iterations and comprehensions*: Write code to build a new list containing the square roots of all the numbers in this list: [2, 4, 9, 16, 25]. Code this as a for loop first, then as a map call, then as a list comprehension, and finally as a generator expression. Use the `sqrt` function in the built-in `math` module to do the calculation (i.e., import `math` and say `math.sqrt(X)`). Of the four, which approach do you like best?
10. *Timing tools*: In [Chapter 5](#), we saw three ways to compute square roots: `math.sqrt(X)`, `X ** .5`, and `pow(X, .5)`. If your programs run a lot of these, their relative performance might become important. To see which is quickest, use the `timer2.py` module ([Example 21-7](#)) we wrote in this chapter to time each of these three tools. Use its `bestoftotal` function to test. Which of the three square root tools seems to run fastest on your device and Python in general? Finally, how might you use `timer2.py` to interactively time the speed of dictionary comprehensions versus for loops? What about comprehensions with if clauses and nested for loops?
11. *Recursive functions*: Write a simple recursion function named `countdown` that prints numbers as it counts down to zero. For example, a call `countdown(5)` will print: 5 4 3 2 1 stop. There's no obvious reason to code this with an explicit stack or queue, but what about a nonfunction approach? Would a generator make sense here?
12. *Computing factorials*: Finally, a computer-science classic (but demonstrative nonetheless). We employed the notion of factorials in [Chapter 20](#)'s coverage of permutations: $N!$, computed as $N*(N-1)*(N-2)*\dots*1$. For instance, $6!$ is $6*5*4*3*2*1$, or 720. Code and time four functions that, for a call `fact(N)`, each return $N!$. Code these four functions (1) as a recursive countdown per [Chapter 19](#); (2) using the functional reduce call per [Chapter 19](#); (3) with a simple iterative counter loop per [Chapter 13](#); and (4) using the `math.factorial` library tool per [Chapter 20](#). Use this chapter's `timeit` to time each of your functions. What conclusions can you draw from your results?

Modules and Packages

Modules: The Big Picture

This chapter begins our in-depth look at the Python *module*—the highest-level program organization unit, which packages program code and data for reuse, and provides self-contained namespaces that minimize variable name clashes across your programs. Modules were introduced in [Chapter 3](#), and we’ve been using them more and more since [Chapter 16](#), but this part of the book provides a focused, detailed look at this Python tool.

This first chapter in [Part V](#) reviews module basics, and offers a general look at the role of modules in the overall structure of programs. In the chapters that follow, we’ll dig into the coding details behind that theory. Along the way, we’ll also flesh out module fine points omitted so far—you’ll learn about reloads, the `__name__` and `__all__` attributes, package imports, relative import syntax, namespace packages, the `__getattr__` hook, the `__main__.py` file, and so on. Because modules and classes are really just glorified *namespaces*, this part formalizes namespace concepts as well.

Module Essentials

In simple and concrete terms, modules typically correspond to Python source code *files*. Each file of code is a module automatically, and modules import other modules to use the names they define. Modules might also correspond to extensions coded in external *languages* such as C, Java, or C#, and even to entire *directories* in package imports, which extend the model for nested files. In all their forms, modules are processed with two statements and one tool:

`import`

Lets a client (importer) fetch a module as a whole

`from`

Allows clients to fetch particular names from a module

`importlib.reload`

Provides a way to reload a module’s code without stopping Python

We’ve used imports in prior examples to load both Python standard-library modules, as well as code files that reside in the current directory—the one we are in when launching the REPL or a script file. While straightforward on the surface, these tools imply an underlying model that’s richer than you

might think. Before we delve into its details, though, let's begin by getting a handle on the purpose of modules in our Python programs.

Why Use Modules?

In short, modules provide an easy way to organize components into a system, by serving as self-contained packages of variables known as *namespaces*. All the names defined at the top level of a module file become attributes of the imported module object, but a file's names can't be seen without imports, and don't clash with names in other files.

This model is related to the scopes we studied in the last part of this book. As we learned in [Chapter 17](#), imports give access to names in a module's *global scope*. The file surrounding a function is always that function's own global scope, but imports allow one file to see the global names of another file as attributes. That is, the module file's global scope *morphs* into the module object's attribute namespace when it is imported.

Ultimately, this allows us to link individual files into a larger program system, with three primary benefits:

Code reuse

Modules make code permanent. Because a module's code is saved in a file, you can both run it multiple times and use it in multiple programs. As you've learned, code you type at a Python REPL goes away when you exit Python, but code in module files can be *rerun* as many times as you wish. Moreover, the tools you define in modules may be *reused* by any number of external clients, and in programs coded both now and in the future.

Minimizing redundancy

Module reuse naturally enables shared copies of common code. If more than one file uses the same or similar code, you can write it once in a module that can then be imported by many clients. For example, in the prior chapter's benchmarking examples, test scripts imported common timer and test-runner functions, instead of repeating them. In other words, modules—like functions—help us factor code to avoid the *redundancy* that results from copy-and-paste programming. As for functions, this minimizes work when common code must be changed.

Namespace partitioning

Modules are also the highest-level namespace structure in a Python program. Although they are fundamentally just packages of names, these packages are also *self-contained*—you can never see a name in another file unless you explicitly import that file. Much like the local scopes of functions, this helps avoid name clashes across your programs. In fact, you can't avoid this feature—everything “lives” in a module. Because both the code you run and the objects you create are always implicitly enclosed in modules, modules group components by nature.

At least that's the abstract story. To truly understand the role of modules in a Python system, we need to digress for a moment and explore the general structure of a Python program.

Python Program Architecture

So far in this book, many examples have sugarcoated the complexity of Python programs. In practice, programs usually involve more than just one file. For all but the simplest scripts, your programs will take the form of *multifile* systems—as the code-benchmarking programs of the preceding chapter

illustrated. Even if you can get by with coding a single file yourself, you will almost certainly wind up using external files that someone else has already written.

This section reviews the general *architecture* of Python programs—the way you divide a program into a collection of source code files (a.k.a. modules) and link the parts into a whole. As you’ll see, Python fosters a *modular* program structure that groups functionality into coherent and reusable units, in ways that are almost automatic. Along the way, this section also explores the central concepts of Python modules, imports, and object attributes.

How to Structure a Program

At a base level, a Python program consists of text files containing Python *statements*, with one main *top-level* file, and zero or more supplemental files known as *modules*.

Here’s how this works. The top-level (a.k.a. *script*) file contains the main flow of control of your program—this is the file you run to launch your application. The module files are libraries of tools, used to group components used by the top-level file, and possibly elsewhere. Top-level files use tools defined in module files, and modules use tools defined in other modules.

Although they are files of code too, module files generally don’t do anything when run directly; rather, they define tools intended for use in other files. A file *imports* a module to gain access to the tools it defines, which are known as its *attributes*—variable names attached to objects such as functions. Ultimately, we import modules and access their attributes to use their tools.

Imports and Attributes

To make this a bit more concrete, [Figure 22-1](#) sketches the structure of a Python program composed of three files: *a.py*, *b.py*, and *c.py*. The file *a.py* is chosen to be the *top-level* script file; it will be a simple text file of statements, which is executed from top to bottom when launched. The files *b.py* and *c.py* are *modules*; they are simple text files of statements as well, but they are not usually launched directly. Instead, as explained previously, modules are normally imported by other files that wish to use the tools the modules define.

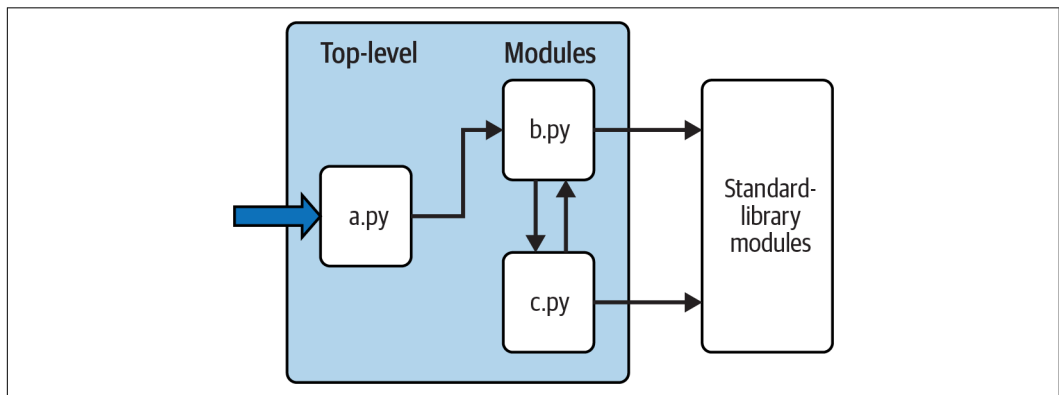


Figure 22-1. Program architecture in Python

For instance, suppose the file *b.py* in [Figure 22-1](#) defines a function called `job` for external use, per [Example 22-1](#) (and ignoring *c.py* for the moment). As you learned when studying functions in [Part IV](#),

b.py will contain a Python `def` statement to generate the function, which you can later run by passing values in parentheses after the function's name.

Example 22-1. *b.py* (module)

```
def job(tool):  
    print(tool, 'coder')
```

This function probably seems trivial at this point in this book, but we're keeping it simple to focus on module basics. Now, suppose *a.py* wants to use `job`. To this end, it might contain Python statements like those in [Example 22-2](#).

Example 22-2. *a.py* (script)

```
import b  
b.job('Python')
```

The first of these, a Python `import` statement, gives the file *a.py* access to everything defined by *top-level code*—that is, code not nested inside a function or class—within the file *b.py*. The code `import b` roughly means:

Load the file *b.py* (unless it's already loaded), and give me access to all its attributes through the name `b`.

To satisfy such goals, `import` (and, as you'll see later, `from`) statements execute and load other files on request. More formally, in Python, cross-file module linking is not resolved until such `import` statements are executed at *runtime*; their net effect is to assign module names—simple variables like `b`—to loaded module *objects*. In fact, the module name used in an `import` statement serves two purposes: it identifies the external *file* to be loaded (by base name `b` here), but it also becomes a *variable* assigned to the loaded module.

Similarly, objects *defined* by a module's code are also created at runtime, while the import is executing: `import` literally runs statements in the target file one at a time to create its contents. Along the way, every name assigned at the top level of the file becomes an attribute of the module, accessible to importers. For example, the second of the statements in *a.py* calls the function `job` defined in the module `b`—and created by running its `def` statement during the import—using object attribute notation. The code `b.job` means:

Fetch the value of the name `job` that lives within the object `b`.

This happens to be a callable function in our example, so we pass a string in parentheses (`'Python'`). If you run *a.py*, the words “Python coder” will be printed—hardly a shocker if you've read prior chapters, but illustrative.

As we've seen, the *object.attribute* notation is general and pervasive in Python code because most objects have useful attributes that are fetched with the “.” operator. Some attributes reference callable objects like functions that take action (e.g., a salary computer), while others are simple values that denote data (e.g., a person's name).

Imports are similarly general because any file can import tools from any other file. For instance, the file *a.py* in [Figure 22-1](#) may import *b.py* to call its function, but *b.py* might also import *c.py* to leverage different tools defined there. In fact, import chains can go as deep as you like: in this example, module *a* can import *b*, which can import *c*, which can import *b* again, and so on. More realistically, in the benchmarking code of the prior chapter, test scripts imported runner modules, which imported timer modules.

The main point behind all this is that modules (and module packages, described in [Chapter 24](#)) are the topmost level of *code reuse* in Python. Components coded in module files can be used both in your original program and in any other programs you may write later. For instance, if we later discover that the function *b.job* in [Example 22-1](#) is widely useful, we can deploy it in completely different programs; all we have to do is import *b* again from the other programs. While unlikely in this simple demo, modules by nature create packages of reusable tools.

Standard-Library Modules

Notice the rightmost portion of [Figure 22-1](#). As we’ve seen along the way, some of the modules that your programs will import are provided by Python itself and are not files you will code.

Python automatically comes with a large collection of utility modules known as the *standard library*. This collection, hundreds of modules large at last count, contains platform-independent support for common programming tasks: operating system interfaces, object persistence, text pattern matching, network and internet scripting, GUI construction, multithreading, and much more.

None of these tools are part of the Python *language* itself, but you can use them by simply importing the appropriate modules on any standard Python installation. Because they are standard-library modules, you can also be reasonably sure that they will be available and will work portably on most platforms on which you will run Python code.

This book’s examples employ a few of the standard library’s modules—*time*, *timeit*, *sys*, and *os* in the last chapter’s code, for instance—but we’ll really only scratch the surface of the library’s story here. For a complete look, browse the Python standard-library reference manual, available online at python.org and elsewhere. See [Chapter 15](#) for more on these manuals; the *PyDoc* tool discussed there also provides library-module info and lists every importable module, including the standard library.

Because there are so many standard-library modules, browsing is the best way to get a feel for what tools are available. You can also find tutorials on Python library tools in books that cover application-level programming, but the standard manuals are free, viewable in any web browser, and updated each time Python is rereleased. As of Python 3.10, `sys.stdlib_module_names` also provides a simple list of all standard-library modules—importable or not:

```
$ python3
>>> len(sys.stdlib_module_names)    # That's a lot of modules
300
```

How Imports Work

The prior section talked about importing modules without fully explaining what happens when you do so. Because imports are at the heart of program structure in Python, this section goes into more formal detail on the import operation to make this process less abstract.

Some C programmers like to compare the Python module import operation to a C `#include`, but they really shouldn't—in Python, imports are not textual insertions of one file into another. They are really *runtime* operations that perform three distinct steps the first time a program imports a given file:

1. *Find* the module's file.
2. *Compile* it to bytecode (if needed).
3. *Run* the module's code to build the objects it defines.

To help you better understand module imports, the following sections explore each of these steps in turn.

First, though, bear in mind that all three of these steps are carried out only the *first time* a module is imported during a program's execution. Later imports of the same module in a program's run bypass all three of these steps and simply fetch the already loaded module in memory.

Python does this by storing loaded modules in a normal dictionary named `sys.modules`, and checking for a module's name there at the start of an import operation. In fact, `sys.modules['name']` means the same as *name* after an `import name`, and `sys.modules`' keys iterator or keys method lists all imported modules:

```
>>> import sys, os
>>> sys.modules['os'] is os           # Name string => module
True
>>> sorted(sys.modules)               # Or sys.modules.keys()
...names of all loaded modules...
```

We'll explore other roles for `sys.modules` in upcoming chapters. On imports, though, if the requested module is not already present in `sys.modules`, a three-step process begins.

Step 1: Find It

First, Python must locate the module file referenced by an `import` statement. Notice that the `import` statement in the prior section's example names the file without a `.py` extension and without its directory path: it just says `import b`, instead of something like `import c:\dir\b.py` or similar on Unix. Path and extension details are omitted in imports on purpose; instead, Python uses a standard *module search path* along with known file types to locate the module file corresponding to an `import` statement.

Because this is the main part of the import operation that programmers must know about, we'll return to this topic by itself in a moment.

Step 2: Compile It (Maybe)

After finding a source code file that matches an `import` statement by traversing the module search path, Python next compiles it to a lower-level form known as *bytecode*, if necessary. We discussed bytecode in [Chapter 2](#), but it's a bit richer than explained there.

When you first import a module, Python compiles the module's `.py` source code file to bytecode, and saves the bytecode in a file with a `.pyc` extension if possible. On later program runs, Python will load the bytecode from its `.pyc` file and skip the compile step, as long as the bytecode file uses a compatible format and was made by the importing Python, and you have not edited and saved the source code file since the bytecode file was made.

Importantly, if you change a module's source code, its bytecode file will be re-created the next time you run a program that imports the module. This ensures that a module's bytecode is always in sync with its source, and your Python.

All of this is automatic and can generally be taken on faith by most Python users, but a brief look at the complete story can help make the process less mysterious than it should be. In more detail, bytecode files can be created in two flavors, the first of which is used by default, and the second of which came online in Python 3.7:

- *Timestamp-based* bytecode files are created by simply importing modules and are the default, original, and most common option. This is the flavor to use if you want imports to be fast, and don't have atypical needs.
- *Hash-based* bytecode files are created by using the `compileall` or `py_compile` library modules. Once created, the Python command-line switch `--check-hash-based-pycs` may be used to configure their operation (see Python's manual for this flag's three options).

In either model, Python automatically saves enough info to know when a bytecode file must be re-created. Specifically, bytecode files embed a “*magic*” number identifying the bytecode's format, along with either the source code file's last-modified *timestamp* and *size*, or a *hash* value derived from the source code file's content. Bytecode filenames also include the implementation name and version of the *Python* that created them.

When a module is imported, Python first checks to see if it was previously imported by the program, and uses the already loaded module if so. Otherwise, it looks for a usable bytecode file corresponding to the module's source code file, by comparing the info saved with the bytecode file against the current specs of the running Python and the source code file. A *.pyc* bytecode file is usable if it has the same base name as the source code file, and:

- Uses a *compatible* format—by checking the “magic” number
- Is *up to date* with the source code file—by comparing either saved timestamp and size, or hash value
- Was created by the running *Python*—by inspecting implementation and version tags in the filename

If there is a bytecode file that passes all these checks, it is loaded, and the compilation step is skipped. If not, the source code is compiled to bytecode, and either saved or resaved in a bytecode file with all the noted info.

Programs still run if bytecode files cannot be saved (compiled code is then simply created in memory and discarded on exit), and the `-B` Python command-line switch can turn off bytecode saves, though it's rarely needed. When they are saved, bytecode files are written to and loaded from a subdirectory named `__pycache__` that's located alongside their corresponding source code files.

The `__pycache__` subdirectory avoids both clutter in your source code folders, and contention and recompiles when multiple Pythons are installed. For example, here's what happens when the same module file in this chapter's examples folder is imported by three different Pythons—two CPythons, and the PyPy we used in the preceding chapter (on Windows, use `dir` and `py` instead of `ls` and `Python` commands here):

```

$ ls
codefile.py

$ python3.12
>>> import codefile

$ python3.8
>>> import codefile

$ pypy3
>>>> import codefile

$ ls
__pycache__  codefile.py
$ ls __pycache__
codefile.cpython-312.pyc  codefile.cpython-38.pyc  codefile.pypy310.pyc

```

Technically, the `__pycache__` subdirectory was introduced in Python 3.2 and isn't available earlier, but this book is focused on Python 3.X only, and you're very unlikely to come across a 3.1 or 3.0 in the wild today.

Also, keep in mind that bytecode compilation happens when a file is being *imported*. Because of this, you will not usually see a `.pyc` bytecode file for the *top-level* file of your program, unless it is also imported elsewhere—only imported files leave behind `.pyc` files on your machine. The bytecode of top-level files is used internally and discarded; bytecode of imported files is saved in files to speed up future imports.

Top-level files are often designed to be executed directly and not imported at all. Later, though, you'll see that it is possible to design a file that serves *both* as the top-level code of a program and as a module of tools to be imported. Such a file may be either executed and imported and does generate a `.pyc` in the latter role. To learn how this works, watch for the discussion of the special `__name__` attribute and `__main__` in [Chapter 25](#).



Running from bytecode only: As a now-special case, programs will also run if Python finds *only* bytecode files but no source code files, though this involves more than just deleting your `.py` files. As of Python 3.2, it generally requires that `m.pyc` files be located where `m.py` files normally would be—via either moving and renaming from `__pycache__` or generating with the legacy option of Python's `compileall` (`-b` in its command-line mode). For instance, the following makes `.pyc` files from all `.py` files in the current directory without requiring moves and renames:

```
$ python3 -m compileall -b -l .
```

Running from just bytecode requires a compatible Python and doesn't fully conceal your code, but is used by some tools to ship programs in standalone form when the hosting Python version can be included or ensured. See Python docs for more info, and the related solution in [“Part I, Getting Started”](#) on page 1157 in [Appendix B](#).

Step 3: Run It

After compiling or loading the module's bytecode, the final step of an import operation executes the bytecode. This effectively runs all the statements in the module's file in turn, from top to bottom, and any assignments made to names during this step generate attributes of the resulting module object. This is how the tools defined by the module's code are created. For instance, `def` statements in a file are run at import time to create functions and assign attributes within the module to those functions. The functions can then be called later in the program by the file's importers.

Because this last import step actually runs the file's code, if any top-level code in a module file does real work, you'll see its results at import time. For example, top-level `print` statements in a module show output when the file is imported. Function `def` statements (and `class` statements up later in this book) simply define objects for later use.

As you can see, import operations involve quite a bit of work—they search for files, possibly run a compiler, and run Python code. Because of this, any given module is imported only *once* per process by default. Future imports skip all three import steps and reuse the already loaded module in memory, per the `sys.modules` check noted earlier. If you need to import a file again after it has already been loaded (for example, to support dynamic customizations), you can force the issue with an `importlib.reload` call—a tool we'll study in the next chapter.

Bear in mind that this process is completely *automatic*—it's a side effect of running programs—and most programmers probably won't care about or even notice the mechanics, apart from faster startups due to skipped compile steps. One part of this process is likely to show up on your radar, though, per the next section's elaboration.

The Module Search Path

As mentioned earlier, the part of the import procedure that most programmers *will* need to care about is usually the first “find it” part—locating the file to be imported. Because you may need to tell Python where to look to find files to import, you need to know how to tap into its *module search path* (the set of directories searched) in order to extend it. This section covers the components that make up the search path and describes how to mod it for folders of your own.

Special case: *built-in* modules like `sys`, coded in C and statically linked into Python, as well as *frozen* modules like `os`, optimized for faster startup as of Python 3.11, are always checked first before scanning the module search path and hence take precedence. Given that there are only a few standard-library modules in these categories, we can safely ignore them here and focus on the search used for the vast majority of modules you'll import—including your own. If you're curious, `sys.builtin_module_names` lists the items in the first of these extrasearch categories:

```
>>> sum(1 for m in sys.builtin_module_names if m[0] != '_')      # Count non _X
11
```

Search-Path Components

In many cases, you can rely on the automatic nature of the module search path and won't need to configure this path at all. If you want to be able to import user-defined files across directory boundaries, though, you will need to customize this path. Roughly, Python's module search path is composed of the concatenation of the following components, some of which are preset for you and some of which you can tailor to tell Python where to look:

1. The home directory of the program
2. Directories listed in `PYTHONPATH` (if set)
3. Standard-library directories and files
4. The *site-packages* directory of third-party extensions
5. The directories listed in any *.pth* files (if present)

Ultimately, the concatenation of these five components initializes the built-in `sys.path`—a changeable list of directory-name strings that are searched from first to last for imported files (we’ll revisit this later in this section). The first, third, and fourth components of the search path are defined automatically. The *second* and *fifth* components, though, can be used to extend the path to include your own source code directories. Here’s a rundown on all five:

Home directory (automatic)

Python first looks for the imported file in the code’s “home” directory. The meaning of this entry depends on how you are running the code. When you’re running a *program*, this entry is the directory containing your program’s top-level script file. When you’re working *interactively* in a REPL, this entry is instead the directory in which you are currently working (which is why we’ve used this directory for imported files so far). Code run with Python’s `-c` and `-m` switches used in earlier chapters also uses the current working directory for the home component.

Because this directory is always searched first, if a program is located entirely in a *single* directory, all of its imports will work automatically with no path configuration required. On the other hand, because this directory is searched first, its files will also *override* modules of the same name in directories elsewhere on the path; be careful not to accidentally hide standard-library modules this way if you need them in your program, or use *package* tools you’ll meet later that can partially side-step this issue with nested folders.

Also remember that this home directory pertains only to the search path used to resolve *imports*, and does not impact the current working directory used for relative *filenames* in your script. Pathless files created by a script will still show up where you are when you launch it (specifically, in the folder returned by `os.getcwd()`), not in the script’s home folder at the front of the module search path. Module imports and file access are disjoint ideas.

Any PYTHONPATH directories (configurable)

Next, Python searches all directories listed in your `PYTHONPATH` environment variable setting, from left to right (assuming you have set this at all: it’s not generally preset for you). In brief, `PYTHONPATH` is simply a list of user-defined and platform-specific names of directories that contain Python source code or bytecode files. You can add all the directories from which you wish to be able to import, and Python will extend the module search path to include all the directories your `PYTHONPATH` lists. See [Appendix A](#) for tips on setting this variable.

Because Python searches the home directory first, this setting is only important when importing files *across* directory boundaries—that is, if you need to import a file that is stored in a *different* directory from the file that imports it. You may need to set your `PYTHONPATH` variable once you start writing substantial programs and tools, but when you’re first starting out, as long as you save all your module files in the directory in which you’re working (i.e., the home directory) your imports will work without needing to make this setting.

Standard-library directories (automatic)

Next, Python automatically searches the directories where the standard-library modules are installed on your machine. These include modules coded in Python, and others coded in C. Because these directories are always searched, they normally do not need to be added to your `PYTHONPATH` (or included in path files, ahead).

The standard-library site-packages directory of third-party extensions (automatic)

Next, Python automatically adds the *site-packages* subdirectory of its standard library to the module search path. By convention, this is the place where most third-party extensions are installed, often automatically by Python's `pip` install tool. Because the *site-packages* install directory of such extensions is always part of the module search path, clients can import the modules of these extensions without any path settings.

Any .pth path-file directories (configurable)

Finally, a lesser-used feature of Python allows users to add directories to the module search path by simply listing them, one per line, in a text file whose name ends with a *.pth* suffix (for “path”). These path-configuration files are a somewhat advanced installation-related feature; we won't cover them fully here, but they provide an alternative to `PYTHONPATH` settings.

In short, text files of directory names dropped in an appropriate directory can serve roughly the same role as the `PYTHONPATH` environment variable setting. For instance, a file with *.pth* extension and any name may be placed in the *site-packages* subdirectory of the installed Python's standard library to extend the module search path. To locate this subdirectory inspect `sys.path` for a path ending in *site-packages*, per its coverage ahead.

When such a file is present, Python will add the directories listed on each line of the file, from first to last, near the end of the module search path list—currently after the *site-packages* directory as described here. In fact, Python will collect the directory names in *all* the *.pth* path files it finds and filter out any duplicates and nonexistent entries. Because they are files rather than shell settings, path files can apply to all users of a given Python, instead of just one user or shell, and may be simpler to set up than environment variables in some contexts.

For more details on this feature, consult Python's library manual, and especially its documentation for the standard-library module `site`—this module configures the locations of Python libraries and path files, and its documentation describes the expected locations of path files in general. When getting started, though, you may be better served by setting `PYTHONPATH`, and only if you must import across directories. Path files are used more often by third-party libraries installed in Python's *site-packages*.

All that being said, Python's import mechanism is wildly extensible, and even more convoluted than described here. For example, `PYTHONPATH` entries may also name *ZIP files* treated like read-only folders; `PYTHONHOME` can set standard-library locations; a file named with suffix *.pth* can fully override `sys.path` norms; “virtual environments” made with the module `venv` avoid package version clashes by localizing search paths to install folders; the relative-import “`..`” syntax of [Chapter 24](#) constrains module search in packages; and all this has morphed regularly, and may again.

While this book covers common usage, its description is intentionally narrow for space, durability, and audience. You should consult Python's manuals for more on the imports story if and when unique requirements arise.

Configuring the Search Path

The net effect of all of the foregoing is that both the `PYTHONPATH` and path file components of the search path allow you to tailor the places where imports look for files. The way you set environment variables and where you store path files varies per platform. For instance, on macOS and Windows, you might set `PYTHONPATH` to a list of directories separated by colons and semicolons, respectively, like this:

```
/Users/me/pycode/utilities:/Volumes/ssd/pycode/package1    # Unix paths list
C:\Users\me\pycode\utilities;d:\pycode\package1           # Windows paths list
```

On Unix systems (e.g., macOS, Linux, and Android), an `export` shell command sets environment variables for the shell and anything it runs, and can be coded in startup files like `~/.bash_profile` for permanence. The following on macOS sets the path to enable imports of modules from the code folders of Chapters 21 and 20 (“...” is snipped text):

```
$ pwd
/Users/me/.../LP6E/Chapter22
$ export PYTHONPATH=/Users/me/.../LP6E/Chapter21:/Users/me/.../LP6E/Chapter20

$ python3
>>> import pybench, permute
>>> pybench
<module 'pybench' from '/Users/me/.../LP6E/Chapter21/pybench.py'>
>>> permute
<module 'permute' from '/Users/me/.../LP6E/Chapter20/permute.py'>

>>> permute.permute1([1, 2, 3, 4])
>>> pybench.runner([(100, 5, '[x ** 2 for x in range(2 ** 16)]')])
```

Path syntax like “.” for parent folders also works on `PYTHONPATH` and is interpreted relative to the current working directory—which is not necessarily the home directory of launched programs, if their top-level scripts live elsewhere. Windows consoles use similar commands (e.g., replace `export` with `set`, or use `setx` for permanence), but Windows users generally set environment variables in *Settings*; search for the environment-variables GUI there.

Instead of—or in addition to—`PYTHONPATH`, you might create a text file in your Python install’s *site-packages* folder named with a *.pth* extension (e.g., *mypath.pth*), which looks like this on macOS:

```
/Users/me/pycode/utilities    # Unix .pth paths file
/Volumes/ssd/pycode/package1  # One path per line
```

These settings are analogous on all platforms, but the details vary too widely for fully inclusive coverage here. See [Appendix A](#) for pointers on extending your module search path with `PYTHONPATH` on various platforms.

To see how your Python configures the module search path on your platform, and to find the location of its *site-packages* folder which can host path files, you can always inspect `sys.path`—the topic of the next section.

The `sys.path` List

If you want to see how the module search path is truly configured on your machine, print the built-in `sys.path` list. This list of directory-name strings is the actual module search path within Python; on imports, Python searches each directory in this list from left to right and uses the first file match it finds.

Inspecting the module search path

Python configures `sys.path` at program startup, merging the path components we met earlier. The result is a list of directories searched on each import of a new file. Python exposes this list for multiple reasons. For one thing, it provides a way to *verify* the search-path settings you’ve made—if you don’t see your settings somewhere in this list after restarting Python, you need to recheck your work. On macOS, for example, with these custom settings:

- `PYTHONPATH` set to `/Users/me/pycode1:/Users/me/pycode2`
- A `mypath.pth` path file in the Python install’s *site-packages* that lists `/Users/me/pycode3`

The search path looks like this when inspected in a REPL or printed from a script:

```
>>> import sys
>>> sys.path
['', '/Users/me/pycode1', '/Users/me/pycode2',
 '/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12.zip',
 '/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12',
 '/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/lib-dynload',
 '/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages',
 '/Users/me/pycode3']
```

The empty string at the front means the current directory, and the two custom settings are merged in per the order given earlier. The rest are standard-library folders and files, and the *site-packages* home for third-party extensions.

Changing the module search path

The `sys.path` list also provides a way for scripts to *tailor* their search paths manually at runtime. By modifying the program-wide `sys.path` list, you modify the search path for all future imports made anywhere in a program’s run. Such changes last only for the duration of the single run, however; `PYTHONPATH` and *.pth* files offer more permanent ways to modify the path—the first per user, and the second per installed Python.

On the other hand, some programs really *do* need to change `sys.path`. Scripts that run on web servers, for example, often run as the user “nobody” to limit machine access. Because such scripts cannot usually depend on “nobody” to have set `PYTHONPATH` in any particular way (grammatically incorrect but true), they sometimes set `sys.path` manually to include required source directories, prior to running import statements. A `sys.path.append`, `sys.path.insert`, or other list operation will often suffice, though will endure for a single program run only:

```
>>> sys.path.append('/Users/me/pycode4')    # Extend search path for this run
>>> import module                          # All imports search the new dir last
```

You can mod `sys.path` arbitrarily in code to influence future imports, and this also works in a REPL. Bear in mind, though, that deleting folders may remove access to important tools, and your changes will be discarded when the program or REPL ends. To make path changes span scripts and sessions, use other techniques instead.

Module File Selection

Besides folders, file *types* also factor into imports. As noted earlier, filename extensions (e.g., *.py*) are omitted from import statements by design: Python chooses the first file it can find on the search path

that matches the imported name. This need not, however, be a *.py* source code file or a *.pyc* bytecode file, as the next section explains.

Module sources

Really, imports are the point of interface to a host of *external components*—source code, bytecode, compiled extensions, ZIP files, Java classes, and more. Python automatically selects any type that matches a module’s name. For example, an `import` statement of the form `import b` might today load or resolve to any of the following:

- A *source code* file named *b.py*
- A *bytecode* file in `__pycache__` named *b.cpython-312.pyc* or similar
- A *bytecode* file named *b.pyc* if no *b.py* source code file was located
- A *directory* named *b*, for package imports described in [Chapter 24](#)
- A compiled *built-in* module, coded in C and statically linked into Python when it is built
- A compiled *extension* module (e.g., *b.so* or *b.pyd*) coded in C and dynamically linked on import
- A source or bytecode file embedded in a *ZIP file*, automatically extracted when imported
- An *in-memory* image’s module, for frozen (standalone) executables
- A *Java* class, in the Jython version of Python
- A *.NET* component, in the IronPython version of Python

Most of the items on this list extend imports beyond simple files. To importers, though, differences in the loaded file type are completely irrelevant, both when importing and when fetching module attributes. Saying `import b` gets whatever module *b* is, according to your module search path, and `b.attr` fetches an item in the module, be it a Python variable or a linked-in C function. Some standard-library modules used in this book, for example, are actually coded in C, not Python; because they look just like Python-coded module files, their clients don’t have to care.

Selection priorities

Given all the options listed in the prior section, there is a potential for conflicts. Python will always load the item with a matching name found in the first (leftmost) directory of your module search path, during the left-to-right scan of `sys.path`. But what happens if it finds multiple matching items in the *same* directory? In this case, Python follows a standard picking order, though this order is not guaranteed to stay the same over time or across implementations.

In general, you should not depend on which type of file Python will choose within a given directory—make your module names distinct, or configure your module search path to make your module-selection preferences explicit.

Path Outliers: Standalones and Packages

Finally, while all of the foregoing reflects normal module usage, some tools bend the rules enough to merit a word in closing. For instance, module search paths are not relevant when you run *standalone executables* discussed in [Chapter 2](#), which typically embed bytecode within their runnable files, and run without path settings. This is a delivery option outside the scope of this book, but see [Appendix A](#) for another overview of options in this domain.

In addition, while it's syntactically illegal to include path and extension details in a standard import, *package imports*, covered in [Chapter 24](#), allow import statements to include *part* of the directory path leading to a file as a set of period-separated names. Even so, package imports still rely on the normal module search path to locate the *leftmost* directory in a package path (they are relative to a directory in the search path), and cannot make use of any platform-specific *path syntax* in the import statements (such syntax works only on the search path). As hinted earlier, packages can also use `""` syntax to restrict import searches—but we'll save this story for [Chapter 24](#).

Chapter Summary

In this chapter, we covered the basics of modules and explored the operation of `import` statements. We learned that imports find the designated file on the module search path, compile it to bytecode, and execute all of its statements to generate its contents. We also learned how to configure the search path to be able to import from directories other than the home and standard-library directories, primarily with `PYTHONPATH` settings.

As this chapter also discussed, the import operation and modules are at the heart of program architecture in Python. Larger programs are divided into multiple files, which are linked together at runtime by imports. Imports in turn use the module search path to locate files, and modules define attributes for external use. The net effect divides a program's logic into reusable and self-contained software components.

You'll see what this all means in terms of actual statements and code in the next chapter. Before we move on, though, let's run through the usual chapter quiz.

Test Your Knowledge: Quiz

1. How does a module source code file become a module object?
2. Why might you have to set your `PYTHONPATH` environment variable?
3. Name the five major components of the module search path.
4. Name four file types that Python might load in response to an import operation.
5. What is a module namespace, and what does a module's namespace contain?

Test Your Knowledge: Answers

1. A module's source code file automatically becomes a module object when that module is imported. Technically, the module's source code is run during the import, one statement at a time, and all the names assigned in the process become attributes of the generated module object.
2. You need to set `PYTHONPATH` only to import from directories other than Python's standard library and the "home" directory—the current directory when working interactively, or the directory containing your program's top-level file. In practice, this might be required for nontrivial programs that use libraries of tools.
3. The five major components of the module search path are the top-level script's home directory (the directory containing it), all directories listed in the `PYTHONPATH` environment variable, the standard-library directories, the *site-packages* root directory for third-party extension installs, and

all directories listed in *.pth* path files located in standard places. Of these, programmers can customize PYTHONPATH and *.pth* files.

4. Python might load a source code (*.py*) file, a bytecode (*.pyc*) file, a C extension module, or a directory of the same name for package imports. Imports may also load more exotic things such as ZIP file components, Java classes under the Jython version of Python, .NET components under IronPython, and statically linked C modules that have no files present at all. In fact, with import extensions, imports can load nearly arbitrary items.
5. A module namespace is a self-contained package of variables, which are known as the *attributes* of the module object. A module's namespace contains all the names assigned by code at the top level of the module file (i.e., not nested in `def` or `class` statements). A module's global scope *morphs* into the module object's attributes namespace. A module's namespace may also be altered by assignments from other files that import it, though this is generally frowned upon (see [Chapter 17](#) for more on the downsides of cross-file changes).

Module Coding Basics

Now that we've studied the larger ideas behind modules, let's turn to some examples of modules in action. Although some of the early topics in this chapter will be review for linear readers who have already applied them in previous chapters' demos, even simple modules can quickly lead us to further details that we haven't yet encountered in full, such as nesting, reloads, scopes, and more, which we'll pick up here.

In general, Python modules are easy to *create*; they're just files of Python program code created with a text editor, and require no special syntax. Because Python does all the work of finding and loading modules, they are also easy to *use*; simply import a module or its names, and use the objects they reference. Let's explore both sides of this fence.

Creating Modules

To define a module, simply use your text editor to type Python code into a text file, and save it with a *.py* extension; any such file is automatically considered a Python module. As we've seen, all the names assigned at the top level of the module become its *attributes* (names associated with the module object) and are exported for clients to use—they morph from variable to module object attribute automatically.

For instance, if you type the code in [Example 23-1](#) into a file called *module1.py* and import it, you create a module object with one attribute—the name `printer`, which happens to be a reference to a function object.

Example 23-1. module1.py

```
def printer(x):          # Module attribute
    print(x)
```

Again, this code may seem simplistic if you read the content of this book that precedes this chapter, but our goal here is to strip out the extraneous bits so we can study modules in isolation.

Module Filenames

Before we go on, let's get more formal about module filenames. You can call modules just about anything you like, but module filenames should end in a *.py* suffix if you plan to import them as modules.

The `.py` is technically optional for top-level files that will be run but not imported (i.e., for *scripts*), but adding it in all cases makes your files' types more obvious, may enable Python-specific features in some text editors and file explorers, and allows you to import any of your files in the future (recall that this is one way to run a file).

Because module names become variable names inside a Python program (without the `.py`), they should also follow the normal variable name rules outlined in [Chapter 11](#). For instance, you can create a module file named `if.py`, but you cannot import it because `if` is a reserved word—when you try to run `import if`, you'll get a syntax error. In fact, both the names of module *files* and the names of *directories* used in package imports (discussed in the next chapter) must conform to the rules for variable names presented in [Chapter 11](#); they may, for example, contain only letters, digits, and underscores. Package directories also cannot contain platform-specific syntax such as spaces in their names.

When a module is imported, Python maps the module name to an external filename by adding a directory path from the module search path of the last chapter to the front, and a `.py` or other extension at the end. For instance, a module named `M` ultimately maps to an external file `directory/M.extension` that contains the module's code.

Other Kinds of Modules

As mentioned in the preceding chapter, it is also possible to create a Python module by writing code in an external language such as C, C++, and others (e.g., Java, in the Jython implementation of the language). Such modules are called *extension modules*, and they are generally used to wrap up external libraries for use in Python scripts or optimize parts of a program. When imported by Python code, extension modules look and feel the same as modules coded as Python source code files—they are accessed with import statements, and they provide functions and objects as module attributes. They're also beyond the scope of this book; see Python's standard manuals for more details.

Using Modules

On the other side of the fence, clients can *use* the simple module file we just wrote by running an `import` or `from` statement. Both statements were introduced in [Chapter 3](#), and have been used in earlier examples. They both find, compile, and run a module file's code if it hasn't yet been loaded, per the process covered in the prior chapter. The chief difference is that `import` fetches the module as a *whole*, so you must qualify to fetch its names, whereas `from` fetches (really, copies) specific *names* out of the module.

Let's see what this means in terms of code. All of the following examples wind up calling the `printer` function defined in module file `module1.py` of [Example 23-1](#) but in different ways.

The import Statement

In the first example that follows, the name `module1` serves two different purposes—it identifies an external file to be loaded, and it becomes a variable in the script, which references the module object after the import. In a REPL:

```
>>> import module1                # Get module as a whole (one or more)
>>> module1.printer('Hello world!') # Qualify to get names
Hello world!
```

The `import` statement simply lists one or more names of modules to load, separated by commas. Because it gives a name that refers to the *whole module* object, we must go through the module name to fetch its attributes (e.g., `module1.printer`).

The `from` Statement

By contrast, because `from` copies *specific names* from one file over to another scope, it allows us to use the copied names directly in the script without going through the module (e.g., `printer`):

```
>>> from module1 import printer          # Copy out a variable (one or more)
>>> printer('Hello world!')            # No need to qualify name (and can't!)
Hello world!
```

This form of `from` allows us to list one or more names to be copied out, separated by commas. Here, it has the same effect as the prior example, but because the imported name is copied into the scope where the `from` statement appears, using that name in the script requires less typing—we can use it directly instead of naming the enclosing module. In fact, we must; `from` doesn't assign the name of the module itself, so `module1` is undefined.

As you'll see in more detail later, the `from` statement is really just a minor extension to the `import` statement—it imports the module file as usual (running the full three-step procedure of the preceding chapter), but adds an extra step that copies one or more *names* (not objects) out of the file. The entire file is loaded, but you're given names for more direct access to its parts.

The `from *` Statement

Finally, the next example uses a special form of `from`: when we use a `*` instead of specific names, we get copies of *all names* assigned at the top level of the referenced module. Here again, we can then use the copied name `printer` in our script without going through the module name:

```
>>> from module1 import *                # Copy out _all_ variables
>>> printer('Hello world!')            # Use names unqualified
Hello world!
```

Technically, both `import` and `from` statements invoke the same import operation; the `from *` form simply adds an extra step that copies all the names in the module into the importing scope. It essentially *merges* one module's namespace into another, which again means less typing for us, albeit at the expense of name segregation.

Note that only a single `*` works in this context; you can't use arbitrary pattern matching to select a subset of names (you could in principle by looping through a module's `__dict__` discussed ahead, but it's substantially more work). Also, note the `from *` may not really get “all” names in the module if that module uses special tricks like `_X` naming or an `__all__` list to hide some of its names from this statement, but we'll defer to [Chapter 25](#) for more on such tools.

And that's it—apart from the search-path configurations of the prior chapter, modules really are simple to use. To give you a better understanding of what really happens when you define and use modules, though, let's move on to look at some of their properties in more detail.



OK, *there is a special case*: The `*` form of the `from` statement form described here can be used *only* at the top level of a module file, not within a function (and generates a syntax error there), because this would make it impossible for Python to detect local variables before the function runs. It's rare to see either `import` or `from` inside a function anyhow, and best practice recommends listing all your imports at the top of a module file; it's not required, but makes them easier to spot, and avoids a speed penalty for re-importing modules on every call to a function. The `from *` has other issues enumerated ahead, and may be best limited to one per file, or interactive REPLs.

Imports Happen Only Once

One of the most common questions people seem to ask when they start using modules is, “Why won’t my imports keep working?” They often report that the first import works fine, but later imports during an interactive session (or program run) seem to have no effect. In fact, this is by design—modules are loaded and run on the first `import` or `from`, and only the first. Because importing is an expensive operation, by default Python does it just once per file, per process. Later import operations simply fetch the already loaded module object.

Initialization code

As one consequence, because top-level code in a module file is usually executed only once, you can use it to initialize names once, but allow their state to change. As a demo, consider the file *init.py* in [Example 23-2](#).

Example 23-2. *init.py*

```
print('hello')
flag = 1                                # Initialize variable - just once!
```

In this example, the `print` and `=` statements run the first time the module is imported, and the variable `flag` is initialized at import time (recall from [Chapter 17](#) that the REPL works like another module file here):

```
$ python3
>>> import init                # First import: loads and runs file's code
hello
>>> init.flag                  # Assignment makes an attribute
1
```

Second and later imports don’t rerun the module’s code; they just fetch the already created module object from Python’s internal modules table. Thus, the variable `flag` is not reinitialized:

```
>>> init.flag = 2              # Change attribute in module
>>> import init                # Just fetches already loaded module
>>> init.flag                  # Code wasn't rerun: attribute unchanged
2
```

Of course, sometimes you really *want* a module’s code to be rerun on a subsequent import. You’ll see how to do this with Python’s `reload` function later in this chapter.

Imports Are Runtime Assignments

Just like `def`, `import` and `from` are *executable statements*, not compile-time declarations. They may be nested in `if` tests, to select among module options; appear in function defs, to be loaded only on calls (subject to the preceding note); be used in `try` statements, to provide defaults on errors; and so on. As an abstract example:

```
if sometest:
    from moduleA import name
else:
    from moduleB import name
```

Wherever they appear, imports are not resolved or run until Python reaches them while executing your program. As one upshot, imported modules and names are not available until their associated `import` or `from` statements run.

Changing mutables in modules

Also like `def`, `import` and `from` are *implicit assignments*:

- `import` assigns an entire module object to a single name.
- `from` assigns one or more names to objects of the same names in another module.

Hence, everything you've already learned about assignment applies to module access, too. For instance, names copied with a `from` become references to shared objects; much like function arguments, reassigning a copied name has no effect on the module from which it was copied, but changing a shared *mutable object* through a copied name can also change it in the module from which it was imported. Consider the following file, *share.py*, in [Example 23-3](#).

Example 23-3. share.py

```
x = 1
y = [1, 2]
```

When importing with `from`, we copy *names* to the importer's scope that initially share *objects* referenced by the module's names (again, the REPL serves as the importing module here):

```
$ python3
>>> from share import x, y          # Copy two names out
>>> x = 'hack'                     # Changes local x only
>>> y[0] = 'hack'                   # Changes shared mutable in place
>>> x, y                           # This module's x and y
('hack', [ 'hack', 2])
```

Here, `x` is not a shared mutable object, but `y` is. The names `y` in the importer and the imported modules both reference the same list object, so changing it from one place changes it in the other; continuing the REPL session:

```
>>> import share                    # Get module name (from doesn't)
>>> share.x                         # share's x is not my x
1
>>> share.y                         # But we share a changed mutable
['hack', 2]
```

For more background on this, see [Chapter 6](#). And for a graphical picture of what `from` assignments do with references, flip back to [Figure 18-1](#) (function argument passing), and mentally replace “caller” and “function” with “imported” and “importer.” The effect is the same, except that here we’re dealing with names in modules, not functions. Assignment works the same everywhere in Python.

Cross-file name changes

In the preceding example, the assignment to `x` in the interactive session changed the name `x` in that scope only, not the `x` in the file—there is no link from a name copied with `from` back to the file it came from. To really change a global name in another file, you must use `import`:

```
$ python3
>>> from share import x, y          # Copy two names out
>>> x = 23                          # Changes my x only

>>> import share                    # Get module name
>>> share.x = 23                   # Changes x in other module
```

This phenomenon was introduced in [Chapter 17](#). Because changing variables in other modules like this is a common source of confusion (and often a subpar design choice), we’ll revisit this technique again later in this part of the book. Subtly, the change to `y[0]` in the prior session is different; it changes an *object*, not a name, and the *name* in both modules references the same, changed object. This can be similarly subpar unless all module clients expect it.

import and from Equivalence

Notice in the prior example that we have to execute an `import` statement after the `from` to access the `share` module name at all. `from` only copies names from one module to another; it does not assign the module name itself. It may help to remember that, at least conceptually, a `from` statement like this one:

```
from module import name1, name2    # Copy these two names out (only)
```

is equivalent to this statement sequence:

```
import module                      # Fetch the module object
name1 = module.name1              # Copy names out by assignment
name2 = module.name2
del module                        # Get rid of the module name - here only
```

Like all assignments, the `from` statement creates new variables in the importer, which initially refer to objects of the same names in the imported file. Only the *names* are copied out, though, not the objects they reference, and not the name of the module itself. When we use the `from *` form of this statement (`from module import *`), the equivalence is the same, but all the top-level names in the module are copied over to the importing scope this way.

Importantly, the first step of the `from` runs a normal `import` operation, with all the semantics outlined in the preceding chapter. As a result, the `from` always imports the *entire* module into memory if it has not yet been imported, regardless of how many names it copies out of the file. There is no way to load just part of a module file (e.g., just one function), but because modules are bytecode in standard Python instead of machine code, the performance implications are generally negligible.

Potential Pitfalls of the from Statement

One downside of the `from` statement is that it makes the meaning of a variable more obscure: `name` is less useful to the reader than `module.name`, and may require a search for the `from` that loaded it.

Because of this, some Python users recommend coding `import` instead of `from` most of the time. Like most advice, though, this doesn't always make sense. `from` is commonly and widely used, without dire consequences. Moreover, it's convenient not to have to type a module's name every time you wish to use one of its tools, especially when working in the REPL.

It is true that the `from` statement has the potential to corrupt namespaces, at least in principle—if you use it to import variables that happen to have the same names as existing variables in your scope, your variables will be silently overwritten. This problem doesn't occur with the `import` statement because you must always go through a module's name to get to its contents: `module.attr` will not clash with a variable named `attr` in your scope. As long as you understand that this can happen when using `from`, though, this isn't a concern in practice: assigning a variable with `from` has the same effect as any other assignment in your code.

On the other hand, the `from` statement has legitimate issues when used in conjunction with the `reload` call, as imported names might reference prior versions of objects. Moreover, the `from *` form really *can* trash namespaces and make code difficult to understand, especially when applied to more than one file—in this case, there is no way to tell which module a name came from, short of searching external files. In effect, the `from *` form collapses one namespace into another, and so defeats the namespace partitioning purpose of modules. We'll save demos of these issues for “[Module Gotchas](#)” on page 638 (at the end of this part of the book), and meet tools that can minimize the `from *` damage with data hiding in [Chapter 25](#).

Probably the best real-world advice here is to generally prefer `import` to `from` for simple modules; to explicitly list the variables you want in most `from` statements; and to limit the `from *` form to just one per file, or the REPL's throw-away code. That way, any names not called out by attribute qualification or `from` lists can be assumed to live in the sole module imported by the `from *`. Some care is required when using the `from` statement, but armed with a little knowledge, most programmers find it to be a convenient way to access modules.

When import is required

The only time you really *must* use `import` instead of `from` is when you must use the same name defined in two different modules. For example, imagine that two files define the same name differently, as in this abstract snippet:

```
# M.py
def func():
    ...do something...

# N.py
def func():
    ...do something else...
```

If you must use *both* versions of this name in your program, the `from` statement will fail—you can have only one assignment to the name in your scope:

```
# O.py
from M import func
from N import func          # This overwrites the one we fetched from M
func()                     # Calls N.func only!
```

An `import` will work here, though, because including the name of the enclosing module makes the two names unique:

```
# 0.py
import M, N                                # Get the whole modules, not their names
M.func()                                   # We can call both names now
N.func()                                   # The module names make them unique
```

This case is unusual enough that you're unlikely to encounter it very often in practice. If you do, though, `import` allows you to avoid the name collision. Another way out of this dilemma is using the `as` extension, a renaming tool that we'll cover fully in [Chapter 25](#) but is simple enough to introduce here:

```
# 0.py
from M import func as mfunc                # Rename uniquely with "as"
from N import func as nfunc
mfunc(); nfunc()                          # Call one or the other
```

The `as` extension works in both `import` and `from` as a simple renaming tool (it can also be used to give a shorter synonym for a long module name in `import`); more on this form in [Chapters 24](#) and [25](#).

Module Namespaces

Modules are probably best understood as packages of names—i.e., places to define names you want to make visible to the rest of a system. Technically, modules usually correspond to files, and Python creates a module object to contain all the names assigned in a module file. But in simple terms, modules are just namespaces (places where names are created), and the names that live in a module are its attributes. This section expands on the details behind this model.

How Files Generate Namespaces

We've seen that files *morph* into namespaces, but how does this actually happen? The short answer is that every name that is assigned a value at the top level of a module file (i.e., not nested in a function or class body) becomes an attribute of that module.

For instance, given an assignment statement such as `X = 1` at the top level of a module file *M.py*, the name `X` becomes an attribute of `M`, which we can refer to from outside the module as `M.X`. The name `X` also becomes a global variable to other code inside *M.py*, but we need to firm up the relationship of module loading and scopes to understand why:

- **Module statements run on the first import.** The first time a module is imported anywhere in a system, Python creates an empty module object and executes the statements in the module file one after another, from the top of the file to the bottom.
- **Top-level assignments create module attributes.** During an import, statements at the top level of the file not nested in a `def` or `class` that assign names (e.g., `=`, `def`) create attributes of the module object. Names assigned by these statements are stored in the module's namespace.
- **Module namespaces are dictionaries.** Module namespaces created by imports may be accessed through the built-in `__dict__` dictionary attribute of module objects, and may be inspected with the `dir` function. `dir` is the same as the sorted keys of `__dict__` for modules, but includes inherited names for classes.
- **Modules are a single scope (local is global).** As we saw in [Chapter 17](#), names at the top level of a module follow the same scope rules as names in a function, but the local and global scopes are the same—or, more formally, they follow the *LEGB* scope rule presented in [Chapter 17](#), but without the *L* and *E* lookup layers.

- **Module scopes outlive imports.** A module's global scope becomes an attribute dictionary of a module object after the module has been loaded. Unlike function scopes, where the local namespace exists only while a function call runs, a module file's scope lives on after the import, providing a source of tools to importers.

Here's a demonstration of these ideas. Suppose we create the module file in [Example 23-4](#) with a text editor, and call it *spaces.py*.

Example 23-4. spaces.py

```
print('starting to load...')
import sys
var = 23

def func(): pass

class klass: pass

print('done loading.')
```

The first time this module is imported (or run as a program), Python executes its statements from top to bottom. Some statements create names in the module's namespace as a side effect, but others do actual work while the import is going on. For instance, the two `print` statements in this file execute at import time:

```
>>> import spaces
starting to load...
done loading.
```

Once the module is loaded, its scope becomes an attribute namespace in the module object we get back from `import`. We can then access attributes in this namespace by qualifying them with the name of the enclosing module:

```
>>> spaces.sys
<module 'sys' (built-in)>
>>> spaces.var
23
>>> spaces.func
<function func at 0x101ce7b00>
>>> spaces.klass
<class 'spaces.klass'>
```

Here, `sys`, `var`, `func`, and `klass` were all assigned while the module's statements were being run, so they are attributes after the import. We'll study classes in [Part VI](#), but notice the `sys` attribute—`import` statements *assign* module objects to names, and any type of assignment to a name at the top level of a file generates a module attribute.

Namespace Dictionaries: `__dict__`

In fact, internally, module namespaces are stored as *dictionary* objects. These are just normal dictionaries with all the usual methods. When needed, we can access a module's namespace dictionary through the module's `__dict__` attribute—for instance, to write tools that list module content generically, as we will in [Chapter 25](#). Continuing the prior section's example:

```
>>> spaces.__dict__.keys()
dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__',
 '__file__', '__cached__', '__builtins__', 'sys', 'var', 'func', 'klass'])
```

The names we assigned in the module file become dictionary keys internally, so some of the keys here reflect top-level assignments in our file. The value of `var`, for example, can be had two ways (though the first is normal):

```
>>> spaces.var, spaces.__dict__['var']
(23, 23)
```

Python also adds some useful names in the module's namespace. For instance, `__file__` gives the name of the file from which the module was loaded (useful to find resources bundled with code), and `__name__` gives its name as known to importers (without the `.py` extension and directory path):

```
>>> spaces.__name__, spaces.__file__
('spaces', '/Users/me/.../LP6E/Chapter23/spaces.py')
```

To see just the names your code assigns, filter out double-underscore names as we did in [Chapter 15](#)'s `dir` coverage and [Chapter 17](#)'s built-in scope coverage, but this time with a generator expression. We can also omit keys because dictionaries generate keys automatically, and the `dir` built-in for modules is just the sorted keys list of `__dict__`:

```
>>> list(name for name in spaces.__dict__ if not name.startswith('__'))
['sys', 'var', 'func', 'klass']
>>> dir(spaces) == sorted(spaces.__dict__)
True
```

As another lesser-used alternative, Python's `vars` built-in function simply fetches the `__dict__` of a module passed to it (an option, perhaps, after you've written enough Python code to wear out your keyboard's underscore key?):

```
>>> vars(spaces) == spaces.__dict__
True
>>> dir(spaces) == sorted(vars(spaces))
True
```

See [Chapter 7](#) for other `vars` roles. You'll see similar `__dict__` dictionaries on *class*-related objects in [Part VI](#) too. In all cases, attribute fetch is similar to dictionary indexing, though only the former kicks off *inheritance* in classes.

Attribute Name Qualification

Speaking of attribute fetch, now that you're becoming more familiar with modules, we should firm up the notion of name qualification more formally too. In Python, you can access the attributes of any object that has attributes using the *qualification* (a.k.a. attribute fetch) syntax *object.attribute*.

Qualification is really an expression that returns the value assigned to an attribute name associated with an object. For example, the expression `spaces.sys` in the previous example fetches the value assigned to `sys` in `spaces`. Similarly, if we have a built-in list object `L`, `L.append` returns the `append` method object associated with that list.

It's important to keep in mind that attribute qualification has nothing to do with the *scope* rules we studied in [Chapter 17](#); it's an independent concept. When you use qualification to access names, you give Python an explicit object from which to fetch the specified names. The LEGB scope rule applies only to bare, unqualified names; it may be used for the leftmost name in a qualification path, but later names after dots search specific objects instead.

This distinction may seem blurred by the fact that module scopes morph into attributes at the end of an import, but thereafter, names are part of a module *object*. For reference, here are the full rules for name resolution in Python:

Simple variables

X means search for the name X in the current scopes (following the LEGB rule of [Chapter 17](#)).

Qualification

X.Y means find X in the current scopes, then search for the attribute Y in the object X (not in scopes).

Qualification paths

X.Y.Z means look up the name Y in the object X, then look up Z in the object X.Y.

Generality

Qualification works on all objects with attributes: modules, classes, C extension types, etc.

As noted earlier, in [Part VI](#) you'll see that attribute qualification means a bit more for classes—it's also the place where inheritance happens. In general, though, the rules outlined here apply to all names in Python.

Imports Versus Scopes

As we've seen, it is never possible to access names defined in another module file without first importing that file. That is, you never automatically get to see names in another file, regardless of the structure of imports or function calls in your program. A variable's meaning is always determined by the locations of assignments in your *source code*, and attributes are always requested of an object explicitly.

For example, consider the following two simple modules. The first, *lex1.py* in [Example 23-5](#), defines a variable X global to code in its file only, along with a function that changes the global X in this file.

Example 23-5. lex1.py

```
X = 88                                # My X: global to this file only

def f():
    global X                          # Change this file's X
    X = 99                           # Cannot see names in other modules
```

The second module, *lex2.py* in [Example 23-6](#), defines its own global variable X and imports and calls the function in the first module.

Example 23-6. lex2.py

```
X = 11                                # My X: global to this file only

import lex1                          # Gain access to names in lex1
lex1.f()                             # Sets lex1.X, not this file's X
print(X, lex1.X)
```

When run, *lex1.f* changes the X in *lex1*, not the X in *lex2*. The global scope for *lex1.f* is always the file *enclosing* it, regardless of which module it is ultimately called from:

```
$ python3 lex2.py
11 99
```

In other words, import operations never give upward visibility to code in imported files—an imported file cannot see names in the *importing* file. More formally:

- Functions can never see names in other functions unless they are physically enclosing.
- Module code can never see names in other modules unless they are explicitly imported.

Such behavior is part of the *lexical scoping* notion—in Python, the scopes surrounding a piece of code are completely determined by the code’s physical position in your file. Scopes are never influenced by function calls or module imports. Some languages act differently and provide for *dynamic scoping*, where scopes really may depend on runtime calls. This tends to make code trickier, though, because the meaning of a variable can differ over time. In Python, scopes more simply correspond to the text of your program.

Namespace Nesting

Finally, although imports do not nest namespaces upward, they do in some sense nest downward. That is, although an imported module never has direct access to names in a file that imports it, using attribute qualification paths it is possible to descend into arbitrarily nested modules and access their attributes. For example, consider the next three files. First, *nest3.py* of [Example 23-7](#) defines a single global name and attribute by assignment.

Example 23-7. nest3.py

```
X = 3
```

Next, *nest2.py* of [Example 23-8](#) defines its own *X*, then imports *nest3* and uses name qualification to access the imported module’s attribute.

Example 23-8. nest2.py

```
X = 2
import nest3

print(X, end=' ')      # My global X
print(nest3.X)         # nest3's X
```

And at the top, *nest1.py* of [Example 23-9](#) also defines its own *X*, then imports *nest2*, and fetches attributes in both the first and second files.

Example 23-9. nest1.py

```
X = 1
import nest2

print(X, end=' ')      # My global X
print(nest2.X, end=' ') # nest2's X
print(nest2.nest3.X)   # Nested nest3's X
```

Really, when *nest1* imports *nest2* here, it sets up a two-level namespace nesting. By using the path of names *nest2.nest3.X*, it can descend into *nest3*, which is nested in the imported *nest2*. The net effect is that *nest1* can see the *X*s in all three files, and hence has access to all three global scopes:

```
$ python3 nest1.py
2 3
1 2 3
```

The reverse, however, is not true: `nest3` cannot see names in `nest2`, and `nest2` cannot see names in `nest1`. This example may be easier to grasp if you don't think in terms of namespaces and scopes, but instead focus on the objects involved. Within `nest1`, `nest2` is just a name that refers to an object with attributes, some of which may refer to other objects with attributes (`import` is an assignment). For paths like `nest2.nest3.X`, Python simply evaluates from left to right, fetching attributes from objects along the way.

Note that `nest1` can say `import nest2`, and then `nest2.nest3.X`, but it cannot say `import nest2.nest3`—this syntax invokes something called *package* (directory) imports, the topic we'll take up in the next chapter after we study reloads here. Package imports also create module namespace nesting, but their `import` statements are taken to reflect *directory* trees, not simple file *import* chains.

Reloading Modules

As we've seen, a module's code is run only once per process by default. To force a module's code to be reloaded and rerun, you need to ask Python to do so explicitly by calling the `reload` built-in function, introduced briefly in [Chapter 3](#). In this section, we'll explore how to use reloads to make your systems more dynamic. In a nutshell:

- Imports—via both `import` and `from` statements—load and run a module's code only the first time the module is imported in a process.
- Later imports use the already loaded module object without reloading or rerunning the file's code. This is true even if you resave the module's source code file during the program run.
- The `reload` function forces an already loaded module's code to be reloaded and rerun. Assignments in the file's new code change the existing module object in place.

So why care about reloading modules? In short, REPL testing and customization. When testing code interactively, it may be easier to reload a module you've changed in another window than it is to restart the REPL.

The more grandiose rationale is *dynamic customization*: the `reload` function allows parts of a program to be changed without stopping the whole program. With `reload`, the effects of changes in components can be observed immediately. Reloading doesn't help in every situation, but where it does, it makes for a much shorter development cycle. For instance, imagine a database program that must connect to a server on startup; because program changes or customizations can be tested immediately after reloads, you need to connect only once while debugging. Long-running servers can update themselves this way, too.

Because Python is interpreted (more or less), it already gets rid of the compile/link steps you need to go through to get a C program to run: modules are loaded dynamically when imported by a running program. Reloading offers a further performance advantage by allowing you to also change parts of running programs without stopping.

One note here: our use of `reload` in this book is limited to modules written in Python. Compiled extension modules coded in a language such as C can be dynamically loaded by imports at runtime, too, but they are out of scope here (though most users probably prefer to code customizations in Python anyhow!).

reload Basics

Unlike `import` and `from`:

- `reload` is a function in Python, not a statement.
- `reload` is passed an existing module object, not a string name.
- `reload` lives in a standard-library module and must be imported itself.

Because `reload` expects an object, a module must have been previously imported successfully before you can reload it (and if the import was unsuccessful due to a syntax or other error, you may need to repeat it before you can reload the module). Furthermore, the syntax of `import` statements and `reload` calls differs: as a function, reloads require parentheses, but `import` statements do not. Abstractly, reloading looks like this:

```
import module                                # Initial module import
...use module.attributes...
...change the module file...

from importlib import reload                 # Get reload itself
reload(module)                              # Get updated module
...use module.attributes...
```

The typical usage pattern is that you import a module, then change its source code in a text editor, and then reload it. This can occur when working interactively, but also in larger programs that reload periodically.

When you call `reload`, Python rereads the module's code and reruns its top-level statements. Perhaps the most important thing to know about `reload` is that it changes a module object *in place*; it does not delete and re-create the module object. Because of that, every reference to an entire module *object* anywhere in your program is automatically affected by a reload. Here are the details:

- **reload runs a module file's new code in the module's current namespace.** Rerunning a module file's code overwrites its existing namespace, rather than deleting and re-creating it.
- **Top-level assignments in the file replace names with new values.** For instance, rerunning a `def` statement replaces the prior version of the function in the module's namespace by reassigning the function name.
- **Reloads impact all clients that use `import` to fetch modules.** Because clients that use `import` qualify to fetch attributes, they'll find new values in the module object after a reload.
- **Reloads impact future `from` clients only.** Clients that used `from` to fetch attributes in the past won't be affected by a reload; they'll still have references to the old objects fetched before the reload.
- **Reloads apply to a single module only.** You must run them on each module you wish to update unless you use code or tools that apply reloads transitively.

reload Example

To demonstrate, here's a more concrete example of `reload` in action. In the following, we'll change and reload a module file without stopping the interactive Python session. Reloads are useful in other scenarios, too, but we'll keep things simple for illustration here. First, in the text editor of your choice, write a module file named *changer.py* with the contents given in [Example 23-10](#).

Example 23-10. *changer.py* (start)

```
message = 'First version'
def printer():
    print(message)
```

This module creates and exports two names—one bound to a string, and another to a function. Now, start the Python interpreter (i.e., your local REPL), import the module, and call the function it exports. The function will print the value of the global `message` variable as you probably expect:

```
$ python3
>>> import changer
>>> changer.printer()
First version
```

Keeping the interpreter active, now edit and save the module file in another window. Change the global `message` variable, as well as the `printer` function body:

```
message = 'After editing'
def printer():
    print('reloaded:', message)
```

Then, return to the Python window and reload the module to fetch the new code. Notice in the following interaction that importing the module again has no effect; we get the original message, even though the file's been changed. We have to call `reload` in order to get the new version:

```
>>> import changer
>>> changer.printer()                # No effect: uses loaded module
First version

>>> from importlib import reload
>>> reload(changer)                  # Forces new code to load/run
<module 'changer' from '/.../LP6E/Chapter23/changer.py'>

>>> changer.printer()                # Runs the new version now
reloaded: After editing
```

Notice that `reload` actually *returns* the module object for us—its result is usually ignored, but because expression results are printed at the interactive prompt, Python shows a default `<module ...>` representation.

You can also reload a module by string name using the `sys.modules` dictionary demoed in the prior chapter if you don't have a variable assigned to it by `import`:

```
>>> import sys
>>> reload(sys.modules['changer'])
```

In this case, it's probably easier to run `import changer` to get a handle on the module, but the `sys.modules` scheme may be useful in programs that need to reload modules more generically. It's also possible to *delete* modules from `sys.modules` to force a reload, but this is generally discouraged for reasons we'll skip here; use `reload`.

reload Odds and Ends

Finally, four brief module-reload notes in closing:

- If you use `reload`, you'll probably want to pair it with `import` instead of `from`, as names fetched with the latter are not updated by reload operations—leaving your names in a state that's strange enough to warrant postponing elaboration until this part's “gotchas” at the end of [Chapter 25](#).
- By itself, `reload` updates only a *single* module, but it's straightforward to code a function that applies it transitively to related modules—an extension we'll save for a case study near the end of [Chapter 25](#).
- Some development tools (e.g. Jupyter notebooks) have REPLs that offer an *auto-reload* mode that may obviate manual `reload` calls. This works only in specific tools, though, and doesn't address customization roles.
- And last, `reload` has had a long and checkered past—morphing from built-in function, to `imp` module attribute in this book's prior edition, to its current `importlib` host. While this may be a symptom of Python's constant morph, which is apt to relocate `reload` again, it must be asked: does this function have trouble working with others?!

Chapter Summary

This chapter drilled down into the essentials of module coding tools—the `import` and `from` statements, and the `reload` call. It showed how the `from` statement simply adds an extra step that copies names out of a file after it has been imported, and how `reload` forces a file to be imported again without stopping and restarting Python. This chapter also detailed what happens when imports are nested, explored the way files become module namespaces, and covered some potential pitfalls of the `from` statement.

Although you've already learned enough to handle module files in most programs, the next chapter extends the coverage of the import model by presenting *package imports*—a way for `import` statements to specify part of the directory path leading to the desired module. As you'll find, package imports give us a hierarchy that is useful in larger systems and allow us to break conflicts between same-named modules. Before we move on, though, here's a quick quiz on the concepts presented here.

Test Your Knowledge: Quiz

1. How do you make a module?
2. How is the `from` statement related to the `import` statement?
3. How is the `reload` function related to imports?
4. When must you use `import` instead of `from`?
5. Name three potential pitfalls of the `from` statement.

Test Your Knowledge: Answers

1. To create a module, you simply write a text file containing Python statements; every source code file is automatically a module, and there is no syntax for declaring one. Import operations load module files into module objects in memory. You can also make a module by writing code in an external language like C or Java, but such extension modules are beyond the scope of this book (and most of its readers).
2. The `from` statement imports an entire module, like the `import` statement, but as an extra step, it also copies one or more variables from the imported module into the scope where the `from` appears. This enables you to use the imported names directly (`name`) instead of having to go through the module (`module.name`).
3. By default, a module is imported only once per process. The `reload` function forces a module to be imported again. It is mostly used to pick up new versions of a module's source code during development, and in dynamic customization scenarios where users change part of a system without restarting it.
4. You must use `import` instead of `from` only when you need to access the same name in two different modules. To make the two names unique, qualify with the names of their enclosing modules obtained with `import`. The `as` extension can render `from` usable in this context as well, by renaming imports uniquely.
5. The `from` statement can obscure the meaning of a variable (which module it is defined in), can have problems with the `reload` call (names may reference prior versions of objects), and can corrupt namespaces (it might silently overwrite names you are using in your scope). The `from *` form is worse in most regards—it can corrupt namespaces arbitrarily and obscure the meaning of variables, so it is probably best used sparingly.

Module Packages

So far, when we've imported modules, we've been loading *files*. This represents typical module usage, and it's probably the technique you'll use for most imports you'll code, especially early in your Python career. The module import story, though, is richer than implied up to this point. This chapter extends it to present module *packages*—collections of module files that normally correspond to *folders* (a.k.a. *directories*) on your device. It covers four topics:

- Package imports, which give part of a folder path leading to a file
- Packages themselves, which organize modules into folder bundles
- Package-relative imports, which use dots within a package to limit search
- Namespace packages, which build a package that may span multiple folders

As you'll find, a package import turns a folder on your computer into another Python namespace, with attributes corresponding to the module files and subfolders that the folder contains. As you'll also learn, package imports are sometimes required to resolve ambiguity when multiple program files of the same name are installed on a device.

Packages are a somewhat advanced topic, which many readers can defer until they gain experience with file-based modules. That said, packages provide an easy way to organize code files that avoids same-name conflicts and is employed by many standard-library and third-party tools that you'll be using. While packages, like much in Python, have grown convoluted over time, a basic knowledge of their usage can be beneficial to most Python learners.

Using Packages

To load items in a package folder, you'll use the normal import statements and tools we've already met, but provide a path of names that reflects a path of nested folders. Let's get started with this user-guide side of the packages story.

Package Imports

Coding package imports is straightforward. In all the places where you have been naming a simple *file* in your import operations, you can instead list a *path* of names separated by periods. For instance, this works in an import statement:

```
import dir1.dir2.mod
```

The same goes for from statements:

```
from dir1.dir2.mod import var
```

And this also applies to already-imported items in reload calls:

```
from importlib import reload
reload(dir1.dir2.mod)
```

The “dotted path” in these contexts is assumed to correspond to a path through the *folder hierarchy* on your device, leading to the module file *mod.py*, or other component with basename *mod* (as we’ve seen, it might also be a bytecode file, C extension module, or other). Most importantly here, the preceding paths indicate that on your device there is a directory *dir1*, which has a subdirectory *dir2*, which contains a module file *mod.py* (or similar).

Furthermore, these paths imply that *dir1* resides within some *container* directory *dir0*, which is a part of the normal module search path. In other words, these imports and reloads imply a directory structure that looks something like this on Unix and Windows, respectively:

```
dir0/dir1/dir2/mod.py      # Or mod.pyc, mod.so, etc.
dir0\dir1\dir2\mod.py      # Ditto, on Windows
```

The container directory *dir0* needs to be added to your module search path unless it’s an automatic part of that path, exactly as if *dir1* were a simple module file.

More formally, the leftmost component in a package import path is *relative to* (located in) a directory included in the `sys.path` module search-path list we explored in [Chapter 22](#). From there down, though, the import statements in your script explicitly give the directory paths leading to modules in packages.

The dotted-path syntax of packages is platform neutral, but also reflects the fact that folder paths in import statements become nested objects: `dir1.dir2.mod` traverses three module *objects* after an import. This syntax also explains why Python complains about a file not being a package if you forget to omit the *.py* extension in import statements: it’s taken to mean a package import!

Packages and the Module Search Path

If you use this feature, keep in mind that the directory paths in your import statements can be only *variables* separated by *periods*. You cannot use any platform-specific path syntax in your import statements, such as `C:\dir1`, `/Users/me/dir1`, or `../dir1`—these do not work syntactically. Instead, use any such platform-specific syntax in your module search path settings to name the directories *containing* your packages.

For instance, in the prior example, *dir0*—the directory name you add to your module search path—can be an arbitrarily long and platform-specific directory path leading up to *dir1*. You cannot use an invalid statement like this:

```
import C:\Users\me\mycode\dir1\dir2\mod      # Error: illegal syntax (Windows)
import /Users/me/mycode/dir1/dir2/mod        # Ditto, on Unix
```

But you can add a path like `C:\Users\me\mycode` or `/Users/me/mycode` to either your `PYTHONPATH` environment variable, a strategically placed `.pth` file, or `sys.path` itself in manual code, and then say this in your script:

```
import dir1.dir2.mod                                # OK: variables and periods
```

In effect, entries on the module search path provide platform-specific directory *prefixes*, which lead to the leftmost names of package paths in `import` and `from` statements. These import statements themselves provide the remainder of the directory path in a platform-neutral fashion.

As for simple file imports, you do not need to add the *container* directory to your module search path if it's already there. Per [Chapter 22](#), the search path automatically includes the “home” directory (where you're working in a REPL, or the container of a launched program's top-level file), along with the standard library's containers, and the *site-packages* third-party install root. While none of these components require search-path mods, your module search path must include all the directories containing *leftmost* components in your code's package-import statements.

Creating Packages

To make packages of your own, you'll bundle module files and nested folders in a package folder. A package folder can also optionally contain an `__init__.py` file run on first import, and a `__main__.py` file run when the entire package folder is run as a bundle. The following sections demo what this looks like in code, one step at a time.

Basic Package Structure

In their simplest form, packages are just folders containing normal module files, and perhaps nested subfolders of the same. Let's set one up as a demo. The following uses indentation to represent the folder nesting we'll be using in this and the following sections:

```
dir0/                                     # Container listed on module search path
  dir1/                                   # Package root folder dir1
    mod.py                               # Module file dir1.mod in package
  dir2/                                   # Nested package folder dir1.dir2
    mod.py                               # Module dir1.dir2.mod in package
```

These nested folders can be created in file explorers, or with the following commands on most platforms; use backslashes on Windows, or simply use this Chapter's folder in the examples package, which has prebuilt the structure:

```
$ mkdir dir1
$ mkdir dir1/dir2                          # Use backward slashes on Windows
```

Admin note: the names of package folders, like those of simple module files, must follow the rules for *variable* names because they become variables where imported. See [Chapter 11](#) for a refresher on the constraints, but in short, use letters, digits, and underscores, and avoid reserved words.

Now, add the nested module files listed in Examples [24-1](#) and [24-2](#). Their top-level code runs on first import and creates attributes as usual, and their titles give their paths (using just Unix forward-slash separators for brevity).

Example 24-1. *dir1/mod.py*

```
var = 'hack'
print('Loading dir1.mod')
```

Example 24-2. *dir1/dir2/mod.py*

```
var = 'code'
print('Loading dir1.dir2.mod')
```

Using the basic package

To use our module package, import its modules from a REPL or another file just as you would for a simple *.py* file, but use dots to specify the path below the *dir0* folder in the search path—which is the current directory in a REPL:

```
$ python3
>>> import dir1.mod                # Package paths in import
Loading dir1.mod
>>> dir1.mod.var                    # Module code run on import
'hack'

>>> import dir1.dir2.mod            # A further-nested path
Loading dir1.dir2.mod
>>> dir1.dir2.mod.var              # Repeat path to get item at end
'code'
```

As for simple top-level modules, the code of modules nested in a package is run only when first imported:

```
>>> import dir1.mod                # No-op if already imported
>>> import dir1.dir2.mod
```

And you generally must *import* a nested module in order to use its attributes—because modules corresponding to folders don't go back to the filesystem on attribute fetches, it's not enough to import just a root folder:

```
$ python3
>>> import dir1                    # Gets dir1, nothing below it
>>> dir1.dir2.mod.var
AttributeError: module 'dir1' has no attribute 'dir2'

$ python3
>>> import dir1.dir2
>>> dir1.dir2.mod.var
AttributeError: module 'dir1.dir2' has no attribute 'mod'

$ python3
>>> import dir1.dir2.mod            # List full path to target
Loading dir1.dir2.mod
>>> dir1.dir2.mod.var
'code'
```

All this works the same in the *from* statement—which, as we've seen, is really just *import* with an extra copy of names. Again, though, we must list a folder by its path in an import statement to be able to access its contents, and the paths listed in imports are taken literally, not fetched from variables:

```

$ python3
>>> from dir1.mod import var           # Package paths in from
Loading dir1.mod
>>> var                               # Don't repeat path to item
'hack'

>>> from dir1.dir2.mod import var
Loading dir1.dir2.mod
>>> var
'code'

>>> from dir1 import dir2               # Paths taken literally
>>> from dir2 import mod                # Not from variable values
ModuleNotFoundError: No module named 'dir2'

```

One potential advantage of `from` here is that it avoids *repeating* a package path every time an item in a package is used. With `import`, you generally must repeat the full path each time, but `from` lets you code the path just once, and use the simple name fetched from it everywhere; this is especially useful if the package's directory structure later changes (as software is wont to do):

```

>>> import dir1.dir2.mod               # import requires paths
>>> dir1.dir2.mod.var
'code'

>>> from dir1.dir2.mod import var      # from can shorten paths
>>> var
'code'

>>> from dir1.dir2 import mod          # And avoids repeating them
>>> mod.var
'code'

>>> import dir1.dir2.mod as mod        # Though "as" can too
>>> mod.var
'code'

```

As the last example shows, though, the `as` extension introduced in the preceding chapter can shorten up paths the same way as `from`—as can a simple manual reassignment, which the `as` sugarcoats (more on `as` in the next chapter).

Package `__init__.py` Files

If you need to run setup code when a package folder is imported, put it in a file named `__init__.py` and store it in the package's folder. Python will automatically run this file's code the first time the package is imported during a program run (or REPL session). Here's how this augments our package's structure:

```

dir0/
  dir1/
    __init__.py    # Run on first import of dir1
    mod__.py
  dir2/
    __init__.py    # Run on first import of dir1.dir2
    mod.py

```

The new `__init__.py` files are listed in Examples 24-3 and 24-4.

Example 24-3. *dir1/___init__.py*

```
var = 'Python'
print('Running dir1.__init__.py')
```

Example 24-4. *dir1/dir2/___init__.py*

```
var = 3.12
print('Running dir1.dir2.__init__.py')
```

Using the updated package

These special `___init__.py` files are optional in each folder of a package. Because they are run on the first import of or through a folder level, though, they provide a natural hook for kicking off package-specific initializations (hence their abbreviated names). In fact, their assignments serve to initialize the *namespace* that corresponds to a folder on your device—as for files, they create attributes of the package’s module object:

```
$ python3
>>> import dir1.dir2.mod                # Runs all ___init__.py files
Running dir1.__init__.py
Running dir1.dir2.__init__.py
Loading dir1.dir2.mod
>>> dir1.var                            # Name in ___init__.py's namespace
'Python'
>>> dir1.dir2.var
3.12
>>> dir1.dir2.mod.var                  # Name in nested mod.py namespace
'code'
```

Technically speaking, packages with `___init__.py` files are called “regular” packages, and those without them are called “namespace” packages, and the demo was a single-folder “namespace” package until adding `___init__.py` made it “regular.” We’ll get into the details behind this distinction later, but apart from the fact that “regular” packages have precedence during module search, the difference is mostly a historical artifact today, and won’t matter in most roles. Of course, `___init__.py` files can also do more than print beacons; we’ll also explore their roles later in this chapter.

As noted earlier, `reload` works with package paths too, if you’ve already imported the paths in question. Continuing the prior REPL session:

```
>>> from importlib import reload
>>> reload(dir1)
Running dir1.__init__.py
<module 'dir1' from '/.../LP6E/Chapter24/dir1/___init__.py'>

>>> reload(dir1.dir2)
Running dir1.dir2.__init__.py
<module 'dir1.dir2' from '/.../LP6E/Chapter24/dir1/dir2/___init__.py'>

>>> reload(dir1.dir2.mod)
Loading dir1.dir2.mod
<module 'dir1.dir2.mod' from '/.../LP6E/Chapter24/dir1/dir2/mod.py'>
```

Notice from the initialization prints that this call reloads just the *single* module on the right end of the path. To do better, we’ll code a reloader tool in the next chapter that, given a package root, can load all the items on a package path like this, one at a time; See [“Example: Transitive Module Reloads” on page 630](#).

Package `__main__.py` Files

Finally, if you want a package's users to be able to run the package as though it were a program or script, add `__main__.py` files to each folder where you wish to support this mode; Python will automatically run these files each time their container folder is launched like a program. Here's how this last bit augments our package's structure:

```
dir0/
  dir1/
    __main__.py      # Run whenever dir1 bundle is run
    __init__.py
    mod__.py
    dir2/
      __main__.py    # Run whenever dir1.dir2 bundle is run
      __init__.py
      mod.py
```

The added `__main__.py` files are listed in Examples 24-5 and 24-6; they're simple so we can focus on packages.

Example 24-5. `dir1/__main__.py`

```
print('Executing dir1.__main__.py')
```

Example 24-6. `dir1/dir2/__main__.py`

```
print('Executing dir1.dir2.__main__.py')
```

Using the updated package

When present, `__main__.py` files are automatically run for a variety of launch options, including direct console command lines that name the containing package folder—which need not be on the module search path in this mode:

```
$ python3 dir1                                # Runs __main__.py (not __init__.py)
Executing dir1.__main__.py
$ python3 dir1/dir2
Executing dir1.dir2.__main__.py
$ python3 dir1/dir2/mod.py                    # Runs nested mod.py
Loading dir1.dir2.mod
```

Package `__main__.py` files are also run for the Python `-m` mode, which, as we've seen, locates an item on the module search path and runs it as a top-level script. Unlike direct console command lines, this mode kicks off the full package-import machinery and runs any `__init__.py` files along the path, but requires the package to be on the search path:

```
$ python3 -m dir1                             # Runs both __init__.py and __main__.py
Running dir1.__init__.py
Executing dir1.__main__.py
$ python3 -m dir1.dir2                        # And package must be on search path
Running dir1.__init__.py
Running dir1.dir2.__init__.py
Executing dir1.dir2.__main__.py
$ python3 -m dir1.dir2.mod                    # Runs mod.py, and parents' __init__.py
Running dir1.__init__.py
Running dir1.dir2.__init__.py
Loading dir1.dir2.mod
```

This is similar in spirit to app “bundles” on macOS and smartphones, which are really folders but can be run as an executable item. The roles for `__main__.py` files are many, but they’re often used to provide a command-line interface to a package of tools. This way, you can import the package to use its tools in another program, but also launch it as a whole to use the tools in a standalone program. This file might also be used to host test code for the package.

A `__main__.py` file is also run if located in a *ZIP file* on the search path. As noted earlier, ZIP files are treated like a normal folder if encountered during a module search, but see Python’s docs for more on this option.

One subtlety here: intrapackage (same-package) imports in `__main__.py` may need to use the package-relative `from .` syntax we’ll study ahead when run by the `-m` argument only. This syntax fails for launches from direct command lines, so you may need to choose a launch mode to support when a `__main__.py` requires same-package imports. As you’ll find, code that wants to support both package and program usage may solve this dilemma with full-path imports.

Why Packages?

Now that you’ve seen how to use and create packages, you might be wondering why in the world anyone would go to all the bother. It’s a valid question. In truth, and despite what you may have heard, packages are completely optional: they are probably overkill early in your coding journey, and even later, you can write amazing programs without them.

In general, though, packages are useful to know about because they help make names unique in both larger programs and libraries published for others to use. By organizing your code as a package folder, you can be fairly sure that the names of its files won’t clash with those of other software on hosting devices. This is the same namespace-segregation role that file-based modules and local scopes in functions play, but extended to the filesystem: because references to your package are qualified by the name of the package’s folder, there’s less risk of same-name collisions.

In addition, packages can make imports more descriptive, ease the task of tracking down the locations of variables in code files, and simplify your module search-path settings. The only time package imports are actually *required*, however, is to resolve ambiguities that may arise when multiple programs with same-named files are installed on the same machine. This is something of an install issue, but it can also become a concern in general practice—especially given the tendency of developers to use simple and similar names for module files.

To help you better understand why all these benefits matter, let’s take a brief detour into the land of the abstract.

A Tale of Two Systems

To illustrate the roles of packages, suppose that a programmer develops a Python program that contains a file called *utilities.py* for common utility code, and a top-level file named *main.py* that users launch to start the program. All over this program, its files say `import utilities` to load and use the common code. When the program is installed, it unpacks all its files into a single directory named *system1* on the target machine that looks like this:

```

system1/
  utilities.py    # Common utility functions, classes
  main.py        # Launch this to start the program
  other.py       # Import utilities to load my tools

```

Now, suppose that a second programmer develops a different program with files also called *utilities.py* and *main.py*, and again uses `import utilities` throughout the program to load its own common code file. When this second system is fetched and installed on the same computer as the first system, its files will unpack into a new directory called *system2* on the host—ensuring that they do not overwrite same-named files from the first system:

```

system2/
  utilities.py    # Common utilities
  main.py        # Launch this to run
  other.py       # Imports utilities

```

So far, there’s no problem: both systems can coexist and run on the same computer. In fact, you won’t even need to configure the module search path to use these programs on your computer—because Python always searches the home directory first (that is, the directory containing the top-level file), imports in either system’s files will automatically see all the files in that system’s own directory. For instance, if you run *system1/main.py*, all imports will search *system1* first. Similarly, if you launch *system2/main.py*, *system2* will be searched first instead. Remember, module search path settings are only needed to import across directory boundaries.

However, suppose that after you’ve installed these two programs on your machine, you decide that you’d like to use some of the code in each of the *utilities.py* files in a system of your own. It’s common utility code, after all, and Python code by nature “wants” to be reused. In this case, you’d like to be able to say the following from code that you’re writing in a third directory to load one of the two files:

```

import utilities
utilities.func('hack')

```

Now the problem starts to materialize. To make this work at all, you’ll have to set the module search path to include the directories containing the *utilities.py* files. But which directory do you put first in the path—*system1* or *system2*?

The issue here is the *linear* nature of the `sys.path` search path. It is always scanned from left to right, so no matter how long you ponder this dilemma, you will always get just one *utilities.py*—from the directory listed *first* (leftmost) on the search path. As is, you’ll never be able to import this file from the other directory at all.

You could try changing `sys.path` within your script before each import operation, but that’s both extra work and error-prone. And changing `PYTHONPATH` before each Python program run is too tedious, and won’t allow you to use *both* versions in a single file in an event. By default, you’re stuck.

This is the issue that packages actually fix. Rather than installing programs in independent directories listed on the module search path directly and individually, you can package and install them as *subdirectories* under a common root. For instance, you might organize all the code in this example as an install hierarchy that looks like this:

```

root/
  system1/
    utilities.py
    main.py
    other.py
  system2/
    utilities.py
    main.py
    other.py
  mycode/                # Here or elsewhere
    myfile.py            # Your new code here

```

Now, add just the common *root* directory to your search path. If your code's imports are all relative to this common root, you can import *either* system's utility file with a package import—the enclosing directory name makes the path (and hence, the module reference) unique. In fact, you can import *both* utility files in the same module, if you use an `import` statement and repeat the full path each time you reference the utility modules:

```

import system1.utilities          # Import from one package
import system2.utilities          # Import from another

system1.utilities.function('hack') # And use names from either
system2.utilities.function('code') # Or both!

```

In short, the names of the enclosing directories here make the module references unique.

Note that you have to use `import` instead of `from` with packages only if you need to access the *same* attribute name in two or more paths. If the name of the called function here were different in each path, you could use `from` statements to avoid repeating the full package path whenever you call one of the functions, as described earlier; as also mentioned, the `as` extension in `import` and `from` can be used to provide unique synonyms too.

Technically, in this case, the *mycode* folder doesn't have to be under *root*—just the packages of code from which you will import. Because you never know when your own modules might be useful in other programs, though, placing them under the common root directory, too, may avoid similar name-collision problems in the future.

Importantly, *path configuration* also becomes simple if you're careful to unpack all your Python systems under a common root like this; you'll only need to add the common root directory once. Moreover, both of the two original systems' imports will keep working unchanged. Because their *home* directories are searched first, the addition of the common root on the search path is irrelevant to code in *system1* and *system2*; they can keep saying just `import utilities` and expect to find their own files when run as programs. As you'll see ahead, though, if they are *also* used as packages, they may need to use `from .` package-relative imports (and *main.py* could be `__main__.py`).

Finally, keep in mind that even if you never create packages of your own, you'll probably use standard-library and third-party tools that do. As a random sample of standard-library packages:

```

from email.message import Message          # Email parsing/construction
from tkinter.filedialog import askopenfilename # Portable GUI toolkit
from http.server import CGIHTTPRequestHandler # Web-server utilities (till Python 3.15?)

```

By bundling their code in a package, such tools become more self-contained, and avoid name conflicts. Whether you opt to do the same for your own code is up to you, but the next few sections dig deeper for if and when you opt-in.

The Roles of `__init__.py` Files

Now that we’ve seen the basics and addressed the why of packages, let’s explore some of the details behind their usage. The `__init__.py` files in our opening demo were simple, but these files can contain arbitrary Python code, just like normal module files. Their names are special because their code is run automatically the first time a Python program imports a directory, and thus serves primarily as a hook for performing initialization steps required by the package. These files can also be completely empty, though, and sometimes have additional roles.

Specifically, the `__init__.py` file serves as a hook for package initialization-time actions, generates a module namespace for a directory, declares a directory as a “normal” Python package, and implements the behavior of `from *` statements when used for folders in package imports:

Package initialization

The first time a Python program imports through a directory, it automatically runs all the code in the directory’s `__init__.py` file. Because of that, these files are a natural place to put code to initialize the state required by files in a package. For instance, a package might use its initialization file to create required data files, open connections to databases, and so on. Typically, `__init__.py` files are not meant to be useful if executed directly (that’s what `__main__.py` is for); instead, they are run automatically when a package is first imported for use elsewhere.

Module namespace initialization

As noted earlier, in the package import model, the directory paths in your script become real nested object paths after an import. For instance, after an import in the preceding demo, the expression `dir1.dir2` works and returns a module object whose namespace contains all the names assigned by `dir2`’s `__init__.py` initialization file. The variable assignments in such files provide attribute namespaces for module objects created for folders, which have no real associated module file, and would otherwise have no place to define names.

Package indicator for search

Package `__init__.py` files are also partly present to declare that a directory is a Python package. In this role, these files serve to prevent directories with common names from unintentionally hiding true modules that appear later on the module search path. Without this safeguard, Python might pick a directory that has nothing to do with your code, just because it appears nested in an earlier directory on the search path. As you’ll see later, the generalizations added for namespace packages subsume some of this role algorithmically, by scanning ahead on the search path to find later items before settling on simple folders. Package `__init__.py` files, though, still give a package higher priority than a same-name folder elsewhere on the search path.

*from * statement behavior*

As an advanced feature, a list of name strings named `__all__` at the top level of an `__init__.py` file defines what is exported for the `from *` statement form. Specifically, the `__all__` list in an `__init__.py` file is taken to be the names of nested submodules that should be automatically imported when `from *` is used on the package directory itself. If `__all__` is not set, the `from *` does not automatically load submodules nested in the package directory; instead, it loads just names defined by assignments in the directory’s `__init__.py` file, including any submodules explicitly imported by code in this file. For instance, `from submodule import X` in a directory’s `__init__.py` makes the name `X` available in that directory’s namespace without an `__all__`.

You’ll see an example of the `__all__` list later, and more coverage of it in [Chapter 25](#) where you’ll find that it also serves to declare `from *` exports of simple file-based modules, not just packages, and is part

of the larger topic of *data hiding*. You can also simply leave `__init__.py` files empty to give your package precedence over *namespace* packages during an import search, but to understand why that matters, we need to move ahead.



Classes conflation caution: As a preview, package `__init__.py` files are not the same as the class `__init__` constructor methods you'll meet in the next part of this book. The former are files of code run when imports first step through a package folder in a program run, while the latter are functions called whenever an instance is created. Both have initialization roles and are optional, but they are otherwise very different—despite their names.

Package-Relative Imports

The coverage of package imports so far has focused on importing package files from *outside* the package. Within the package itself, imports of same-package files can use the same full path syntax as imports from outside the package—and as you'll see, sometimes should. However, files within a package can also make use of *intrapackage* syntax that makes imports *relative* to the package itself, and can ensure that imports in a package load the package's own files.

Relative and Absolute Imports

The code behind this is straightforward. Imports run by files used as part of a package can use special syntax like the following, which works only in files used as package components, and only in `from` statements, not `import`:

```
from . import module           # Import a module in this package (only)
from .module import name      # Import a name from a module in this package
from .. import module         # Import a module sibling of the parent folder
from ..module import name     # Import a name from a parent-sibling module
```

This syntax wouldn't make sense in `import`, because that statement assigns modules to simple names, not paths. In a `from`, though, when the source of the import begins with (or is only) dots like this, the import is known as *relative*—it identifies an item relative to the folder of the enclosing package itself.

This name comes from the similarity to relative *filename* paths, which reference a file per the current working directory (CWD), as covered in [Chapter 9](#). Much as in filenames, `"."` means the immediately enclosing package folder, and `".."` means its parent folder another level up (and each additional dot means one level higher, though this is rarely used). Here, though, the effect names an item relative to an importer's package, not a content file relative to the CWD.

Importantly, relative imports within a package search *only* the referenced package: they never check the folders listed in `sys.path` as normal imports do (and skip the built-in and frozen modules precheck). Moreover, relative-import syntax can be used *only* when a file is being utilized as part of a package, and fails otherwise. This has consequences both good and bad that we'll explore in a moment.

Imports in files being used as part of a package can still be coded without dots as usual, in both `import` and `from`:

```
import module                 # Import a module from a folder in sys.path
from module import name       # Import a name from the same
```

Sans leading periods like this, an import is known as *absolute*—it identifies an item that's located in a folder listed in `sys.path`. This is the normal behavior of imports in Python, and there's really nothing

“absolute” about it per the filename analogy (absolute filename paths give a complete filesystem path). In fact, these “absolute” imports are really *relative* to an entry in the module search path, but we’re stuck with the terminology today.

Also importantly, absolute imports within a package do *not* automatically check the package itself: they skip the package and move right to a search of folders on `sys.path`. As we’ve seen, `sys.path` may include the package’s folder anyhow, by virtue of the REPL’s CWD or the home folder of the top-level script, and `PYTHONPATH` or other settings. By themselves, though, absolute imports don’t check the package for imports run in package files.

Relative-Import Rationales and Trade-Offs

So why the dots? In short, relative imports allow a package to ensure that its imports will load its *own* modules. Because relative imports search *only* the package itself, they won’t inadvertently load a same-named but unrelated module elsewhere on the host that happens to be accessible through `sys.path`. This in turn makes the package more self-contained: without relative imports, such an unrelated module might break the package’s code; with them, packages are less reliant on client search-path settings that they cannot predict or control.

The downside is that this model is an *all-or-nothing* proposition. You essentially must choose a mode for your files—package or program. Relative imports give visibility to the package itself, but cannot be used in nonpackage mode, and absolute imports can be used in nonpackage mode, but do not give visibility to the package itself. This combination seems a catch-22 that limits your code’s utility.

That being said, if you’re coding a library of tools, this may be a nonissue: you can adopt relative imports throughout your package, and provide a `__main__.py` file to run the package as a program with the `-m` switch. If you’re writing a more traditional folder of code that you want to use arbitrarily, though, your options appear to be limited.

One easy solution to this dilemma is to simply *avoid* relative imports altogether, and use full, explicit, and “absolute” package-import paths everywhere in your code. That is, use imports that list the path to the desired item from and including the package’s root folder itself:

```
import package.module           # Import a module in this package
from package import module      # Same as: from . import module
from package.module import name # Same as: from .module import name
```

For this to work, the package’s root folder must reside in a folder listed on `sys.path`, but this will be the *normal* case—there’s no way for clients to import the package’s code otherwise. This also doesn’t directly support exotic imports like “..” parent siblings, but these seem likely to be rare (e.g., only one standard-library tool uses them today).

With this nonrelative equivalence, code folders can be used more flexibly, and both direct command lines and the `-m` module-mode switch can be used to launch files in the package as programs. The next section shows how.

Package-Relative Imports in Action

To demo all of the foregoing points, let’s return to the simple package we coded at the start of this chapter. As you’ll recall, its `__main__.py` file is run automatically when the folder is run as a bundle, but it will also play the role of any top-level file in the folder launched as a script. When we last met this file, it simply contained a print; here’s a refresher of the last-known state of the files we’ll be using here so you don’t have to flip back:

```
# prior dir1/mod.py
var = 'hack'
print('Loading dir1.mod')

# prior dir1/__main__.py
print('Executing dir1.__main__.py')
```

The normal-import warm-up

This `__main__.py` worked as advertised, but things get more complicated if a file run as a top-level script tries to import another module in its own package folder—as in the rewrite of [Example 24-7](#).

Example 24-7. `dir1/__main__.py` (modified)

```
import mod
print('Executing dir1.__main__.py:', mod.var.upper())
```

Coded this way, the file can be run with a direct command line (and both as a folder and a script), but *not* with Python’s `-m` module-mode switch, which brings the package machinery online (notice the `__init__.py` output in this mode):

```
$ python3 dir1                                     # Launch with a direct command line
Loading dir1.mod
Executing dir1.__main__.py: HACK
$ python3 dir1/__main__.py                         # As folder bundle or explicit file
Loading dir1.mod
Executing dir1.__main__.py: HACK

$ python3 -m dir1                                  # Launch with Python -m module switch
Running dir1.__init__.py
ModuleNotFoundError: No module named 'mod'
$ python3 -m dir1.__main__                         # As package root or explicit module
Running dir1.__init__.py
ModuleNotFoundError: No module named 'mod'
```

The first two commands in this work because `__main__.py` is run as a normal, nonpackage program, and finds its imported sibling per the “home” entry on `sys.path`, the top-level file’s own folder. The problem with the last two commands is that they use the file as part of a package: the `import mod` in `__main__.py` is then interpreted as an *absolute* import—which *skips* the enclosing package. Hence the fails.

The relative-import adventure

Our first reaction might be to add *relative* imports to appease the package system—as in [Example 24-8](#).

Example 24-8. `dir1/__main__.py` (modified)

```
from . import mod
print('Executing dir1.__main__.py:', mod.var.upper())
```

Coded this way, we’ve fixed `-m` usage, but *broken* direct command lines:


```

$ python3 -m dir1
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK
$ python3 -m dir1.__main__
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK

$ python3 dir1
ImportError: attempted relative import with no known parent package
$ python3 dir1/__main__.py
ImportError: attempted relative import with no known parent package

```

The first two commands in this work because `-m` invokes package behavior, which enables the relative “.” import syntax that searches the package and finds the target module. The problem with the last two commands is that Python doesn’t allow relative imports to be used in nonpackage mode—which dooms these runs from the start, regardless of search-path settings. Hence the fails.

Under this regime, then, it would seem we must *choose* package or nonpackage roles for our code.

The absolute-import solution

As noted, though, an easy way out of this constraint is to use normal package imports that explicitly spell out absolute import paths (which, again, are actually relative to `sys.path`) from the package root—as in [Example 24-9](#).

Example 24-9. `dir1/__main__.py` (modified)

```

import dir1.mod
print('Executing dir1.__main__.py:', dir1.mod.var.upper())

```

As also noted, the package root, `dir1`, will normally be on `sys.path`, because that’s the only way to use its code from outside the package (clients must import with paths that start at the `dir1` package root too).

To demo here, we’ll add the package root explicitly using a relative—in *filesystem* terms—path of “.” for the current directory (in typical use, this might instead be an absolute path, or Python’s automatic *site-packages* root of installed packages). We must set this here because the top-level file’s “home” on the search path in direct-command mode is one level *below* the package root, and both direct-command and `-m` modes need access to the package root in general:

```

$ export PYTHONPATH=. # Or similar outside Unix: see Chapter 22
$ python3 dir1
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK
$ python3 dir1/__main__.py
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK

$ python3 -m dir1 # Both usage modes work, sans "." imports
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK
$ python3 -m dir1.__main__

```

```
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK
```

Now launches by *both* direct command lines and the `-m` switch work, because we're not using a tool that limits the utility of our code to just one mode. The end result is *dual-mode* code—it can be used as both program and package.

If you'd rather not repeat import paths at each item reference, both launch modes also work if we code `__main__.py` in any of these ways (test on your own to verify; this is just import norms at work):

```
import dir1.mod as mod
print('Executing dir1.__main__.py:', mod.var.upper())

from dir1 import mod
print('Executing dir1.__main__.py:', mod.var.upper())

from dir1.mod import var
print('Executing dir1.__main__.py:', var.upper())
```

For more fun, you can also use a `from *` in this scheme, if you add an `__all__` to the package root's initialization file (subject to all the standard disclaimers about the evils of `from *` outlined in [Chapter 23](#)):

```
# dir1/__init__.py
__all__ = ['mod']

# dir1/__main__.py
from dir1 import *
print('Executing dir1.__main__.py:', mod.var.upper())
```

Again, if you're sure that your code will only ever be used as part of a package, you could use relative imports in all of its files to access same-package modules. Moreover, the absolute-path solution arrived at here could be applied *only* for files meant to be launched as top-level *scripts*; other files only imported can use package-relative imports.

In both cases, relative imports come with the advantage that a same-named module earlier on `sys.path` won't accidentally override a crucial module in the package. But they also limit a file to package-only roles; if you want package files to support *both* launches and imports, relative imports are not your best option.

It should also be noted that we're skipping some details here for space. For one, `-m` mode unusually also checks the CWD for absolute imports, making the demo's `PYTHONPATH` setting optional for this mode and demo only. For another, some other resources suggest that relative-import failures in scripts stem from their name `'__main__'`, but files run with `-m` have the same name and employ other protocols that are too obscure to cover here.

Because package-relative imports are both prone to change and mostly meant for programmers building large-scale libraries of code for others to use, this Python-learners text will defer to Python's docs for further details. Modules are feature-rich tools, to say the least, but the good news is that namespace packages—the topic of the next and final section of this chapter—do not add any new syntax, but simply allow a module to span folders. To see how, let's move on.

Namespace Packages

Now that you’ve learned all about package and package-relative imports, there’s one last package-related subject to cover. The evolutionary path of modules in Python eventually led to what are known as *namespace packages*—packages that may be composed of one or more folders located on different parts of the module search path.

While packages split across folders are probably atypical in the wild, the generalizations added to support namespace packages in the import-search *algorithm* (procedure) also allow for packages without `__init__.py` files in general. Since this revised algorithm is now just *the* import search, namespace packages are something of an incidental topic.

To understand namespace packages’ place in the broader modules picture, though, as well as the changes they ushered in, we need a quick history lesson.

Python Import Models

All told, Python has four distinct import schemes. The following enumerates them from original to newest, with representative but incomplete examples (as we’ve seen, `reload` also allows a `*` wildcard and `reload` reloads any module passed to it, but these are just add-on topics):

Basic modules

The original model, used to import files and their contents, relative to the `sys.path` module search path:

```
import module
from module import name
```

Basic packages

The original package model used to import from folder paths relative to the `sys.path` search path, where each package is contained in a single directory that has an `__init__.py` file:

```
import folder.folder.module
from folder.module import name
```

Package-relative imports

The model used for intrapackage (same-package) imports of the prior section, with its relative and absolute lookup rules for imports with and without leading dots, respectively:

```
from . import module
from .module import name
```

Namespace packages

The newest package model, which is still relative to `sys.path`, but allows packages to span multiple directories, and removes the requirement that packages must define `__init__.py` files:

```
import anyfolder.anyfolder.module
from anyfolder import name
```

The first two of these models are self-contained, but the third tightens up the search order and extends syntax for same-package imports, and the fourth upends some of the notions and requirements of the prior package model. In fact, Python now formally defines two flavors of packages:

- The original model, now known as *regular packages*
- The alternative model, known as *namespace packages*

The original and alternative package models are not mutually exclusive and can be used simultaneously in the same program. In fact, the namespace package model works as something of a *fallback option*, recognized only if basic modules and regular packages of the same name are not present on the module search path. Despite their showy title, namespace packages are really just a mutation of import search, as the following sections explain.

Namespace-Package Rationales

First off, it's important to know that a namespace package is not fundamentally different from a regular package: it is just a different way of creating packages. Moreover, they are still relative to `sys.path` at the top level: the leftmost component of a dotted namespace-package path must still be located in an entry on the normal module search path.

In terms of physical structure, though, regular and namespace packages have notable differences. Regular packages have an `__init__.py` file that is run automatically, and they reside in a single directory. Namespace packages, by contrast, cannot contain an `__init__.py`, and may or may not span multiple directories collected at import time. Because the presence of `__init__.py` differentiates package type, *none* of the directories that make up a namespace package can have this file, but the content nested within each of them is treated as a single composite package.

The *rationale* for namespace packages is rooted in package *installation* goals that may seem obscure unless you are responsible for such tasks. In short, though, they resolve a potential for collision of multiple `__init__.py` files when package parts are merged, by removing this file completely. Moreover, by providing standard support for packages that can be split across multiple directories and located in multiple `sys.path` entries, namespace packages both enhance install flexibility and replace multiple incompatible solutions that had arisen to address this goal.

Though split-folder packages have some narrow roles, average Python users will probably find namespace packages' biggest benefit to be that they remove the requirement for package `__init__.py` initialization files, and thus allow any directory of code to be used as an importable package. To see how, let's move on to the nitty-gritty of import search.

The Module Search Algorithm

To understand the way that namespace packages change and extend import search, we have to look under the hood to see how the import operation works.

As we've seen, Python fulfills imports by searching for a name among a set of candidate folders. For the *leftmost* components of imports, the set of candidates is all the directories listed on the `sys.path` module search path. For components of packages either nested in *package* imports or named in *relative* imports, the set of candidates is just the package itself. While package folders have just one instance of a given name, search paths may have many.

The way the import search selects items from these candidates is subtler than implied so far. For each *directory* in an import search's set of candidates, Python tests for a variety of matches to an imported *name*, in the following order (using Unix `/` for folder separators and `{...}` to mean a choice):

1. If `directory/name/__init__.py` is found, a regular package is imported and returned.
2. If `directory/name.{py, pyc, or other module extension}` is found, a simple module is imported and returned.

3. If *directory/name* is found and is a directory, it is recorded and the scan continues with the next directory in the search's set of candidates.
4. If none of the above was found, the scan continues with the next directory in the search's set of candidates.

If this search's candidate scan completes without returning a regular package or module by steps 1 or 2, and at least one directory was recorded by step 3, then a *namespace package* is created and returned. Else an error is reported.

The creation of the namespace package happens immediately and is not deferred until a lower-level import occurs. The new namespace package is a module object that does not have a `__file__` attribute, but has a `__path__` set to an iterable of the directory path strings that were found and recorded during the candidates scan by step 3.

This `__path__` attribute is then used as the set of candidates for later and deeper accesses, to search the one or more component folders of the namespace package. That is, each recorded entry on a namespace package's `__path__` is searched whenever further-nested items are requested, instead of the sole directory of a regular package.

Viewed another way, the `__path__` attribute of a namespace package serves the same role for lower-level components that `sys.path` does at the top for the leftmost component of package import paths; it becomes the “parent” search path for accessing lower items using the same four-step procedure just sketched.

The net result is that a namespace package is a sort of *virtual concatenation* of directories located on one or more search candidates. Once a namespace package is created, though, there is no functional difference between it and a regular package; it supports everything we've learned for regular packages, including package-relative import syntax.

Importantly, because a *single directory* lacking an `__init__.py` but nested in a search-candidate folder is classified as a namespace package by this algorithm's step 3, any such directory qualifies as a package. The only operational difference between such a single folder and a regular package folder is that the former has lower *search precedence*: both same-named folders with an `__init__.py` and simple modules anywhere in a search path are chosen first by steps 1 and 2.

In other words, although the mods made to support namespace packages make `__init__.py` files optional in package folders, adding one, even if empty, ensures that a package will be selected instead of a same-named folder later on a search path (it may also boost import speed by ending search-path scans at step 1, but this is a one-time event). Whether this, or the other roles of `__init__.py` we met earlier, warrants including this file will naturally depend on your goals.

Namespace Packages in Action

Technically, we've already seen namespace packages at work: the opening step of the demo at the start of this chapter made *single-folder* namespace packages because its folders didn't yet have `__init__.py` files. Again, any folder nested in a `sys.path` search-path folder qualifies as a package, as long as it's not hidden by a same-named “regular” package with `__init__.py` or a simple module elsewhere on the path.

To see the grander folder-concatenation effect of “virtual” namespace packages work, let's work through a quick demo. To begin, make two modules in a nested directory structure that has subdirectories named `sub` located in different parent directories, `part1` and `part2`—like this in indentation notation meant to represent nesting:

```

ns/
  part1/
    sub/
      mod1.py
  part2/
    sub/
      mod2.py

```

Note that there are no `__init__.py` files here—as noted earlier, these files cannot be used in namespace packages, as this is their chief differentiation from regular, single-folder packages. We can create this demo’s folders with console commands like the following; translate `/` to `\` and omit the `-p` on Windows, or use a file explorer or the prebuilt folders in the examples package if you prefer:

```

$ mkdir -p ns/part1/sub      # Two subdirs of same name in different dirs
$ mkdir -p ns/part2/sub      # And similar on Windows

```

The modules at the end of these paths are coded in Examples 24-10 and 24-11 to simply print on imports as a trace.

Example 24-10. ns/part1/sub/mod1.py

```
print('Loading ns/part1/sub/mod1')
```

Example 24-11. ns/part2/sub/mod2.py

```
print('Loading ns/part2/sub/mod2')
```

Now, if we add *both* `part1` and `part2` to the module search path, `sub` becomes a namespace package spanning both folders, with the two module files available under that name even though they live in separate physical directories. Here’s the path-setting command on Unix (for Windows, use `set` and `:`, and see [Chapter 22](#) for more tips); this uses paths relative to the CWD, but in practice would more likely list two full, absolute paths instead:

```
$ export PYTHONPATH=ns/part1:ns/part2
```

When imported directly, the namespace package is the *virtual concatenation* of its individual directory components, and allows further nested parts to be accessed through its single, composite name with normal imports (as usual, paths have been shortened here for space with “...” and line breaks were added for fit):

```

$ python3
>>> import sub
>>> sub                                     # Namespace packages: nested search paths
<module 'sub' (namespace) from
['/.../LP6E/Chapter24/ns/part1/sub', '/.../LP6E/Chapter24/ns/part2/sub']>

>>> from sub import mod1
Loading ns/part1/sub/mod1
>>> import sub.mod2                         # Content from two different directories
Loading ns/part2/sub/mod2

>>> mod1
<module 'sub.mod1' from '/.../LP6E/Chapter24/ns/part1/sub/mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from '/.../LP6E/Chapter24/ns/part2/sub/mod2.py'>

```

This split-folder package also works if we import through the namespace package name *immediately*—because the namespace package is made when first reached, the timing of path extensions is irrelevant:

```

$ python3
>>> import sub.mod1
Loading ns/part1/sub/mod1
>>> import sub.mod2                                # One package spanning two directories
Loading ns/part2/sub/mod2

>>> sub.mod1
<module 'sub.mod1' from '/.../LP6E/Chapter24/ns/part1/sub/mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from '/.../LP6E/Chapter24/ns/part2/sub/mod2.py'>

>>> sub
<module 'sub' (namespace) from
['/.../LP6E/Chapter24/ns/part1/sub', '/.../LP6E/Chapter24/ns/part2/sub']>
>>> sub.__path__
_NamespacePath(
['/.../LP6E/Chapter24/ns/part1/sub', '/.../LP6E/Chapter24/ns/part2/sub'])

```

Interestingly, *relative imports* work in namespace packages too, if we add one as in [Example 24-12](#).

Example 24-12. *ns/part1/sub/mod1.py (modified)*

```

from . import mod2
print('Loading ns/part1/sub/mod1')

```

The added package-relative import statement references a file in the package—even though the referenced file resides in a *different directory*:

```

$ python3
>>> import sub.mod1                                # Relative import of mod2 in another dir
Loading ns/part2/sub/mod2
Loading ns/part1/sub/mod1
>>> import sub.mod2                                # Already imported by mod1.py: not rerun
>>> sub.mod2
<module 'sub.mod2' from '/.../LP6E/Chapter24/ns/part2/sub/mod2.py'>

```

As you can see, namespace packages are like ordinary single-directory packages in every way, except for having a split physical storage. By extension, this is why code folders without `__init__.py` files are exactly like regular packages, but with no initialization logic to be run; they’re just an instance of namespace packages with a single directory.

Like package-relative imports, this book is also going to defer to Python’s manuals for more details on this subject. Namespace packages are a potentially useful tool, but most Python learners are probably better served by first mastering packages that map to real folders, before tackling those that are spread virtually across a host’s directories.

Chapter Summary

This chapter introduced Python’s *package import* model—an optional but useful way to explicitly list part of the directory path leading up to modules. Package imports are still relative to a directory on your module search path, but your script gives the rest of the path to the module explicitly. Packages may be built with a simple folder and can make imports more meaningful, simplify import search-path settings, and resolve ambiguities when there is more than one module of the same name—the name of the enclosing directory makes them unique.

Because it's relevant only to code in packages, we also explored *relative imports*: a way for imports in package files to select modules in the same package explicitly using leading dots in `from`, instead of relying on a search in the host. Finally, we surveyed *namespace packages*: a tool that allows a package to span multiple directories as a fallback option of import searches, and make initialization files optional in single-folder packages.

The next chapter surveys a handful of both common and advanced module-related topics, such as the `__name__` usage mode variable, the `__getattr__` attribute hook, and name-string imports, and codes useful module tools along the way. As usual, though, let's close out this chapter first with a short quiz to review what you've learned here.

Test Your Knowledge: Quiz

1. What is the purpose of an `__init__.py` file in a module package directory?
2. How can you avoid repeating the full package path every time you reference a package's content?
3. Which directories require `__init__.py` files?
4. When must you use `import` instead of `from` with packages?
5. What is the difference between `from pkg import name` and `from . import name`?
6. What is a namespace package?

Test Your Knowledge: Answers

1. The `__init__.py` file serves to declare and initialize a regular module package; Python automatically runs its code the first time you import through a directory in a process. Its assigned variables become the attributes of the module object created in memory to correspond to that directory. It's optional for package folders but gives a folder search precedence over other folders of the same name that don't have this file.
2. Use the `from` statement with a package to copy names out of the package directly, or use the `as` extension with the `import` statement to rename the path to a shorter synonym. In both cases, the path is listed in only one place, in the `from` or `import` statement.
3. Trick question! These files used to be required for packages in earlier Pythons but became optional with accommodations for namespace packages in the module search algorithm. As noted in answer 1, though, these folders still have valid, if optional, roles, including boosting a folder's import-search precedence.
4. You must use `import` instead of `from` with packages only if you need to access the *same name* defined in more than one path. With `import`, the path makes the references unique, but `from` allows only one version of any given name (unless you also use the `as` extension to rename uniquely).
5. The `from pkg import name` is an *absolute* import—the search for `pkg` skips an enclosing package and then tries the “absolute” directories in `sys.path`. A statement `from . import name`, on the other hand, is a *relative* import—`name` is looked up relative to the package in which this statement is contained, only.

6. A *namespace package* is an extension to the import model that corresponds to one or more directories that do not have `__init__.py` files. When Python finds these during an import search and does not find a simple module or regular package first, it creates a namespace package that is the virtual concatenation of all found directories having the requested module name. Further nested components are looked up in all the namespace package's directories. The effect is similar to a regular package, but content may be split across multiple directories.

Module Odds and Ends

This chapter concludes this part of the book with an assortment of module-related topics—data hiding, the `__future__` module, the `__name__` variable, name-string imports, the `__getattr__` hook, transitive reloads, and more—along with the usual set of gotchas and exercises related to what we’ve covered in this part of the book. Along the way, we’ll build some useful tools that combine functions and modules. Like functions, modules are more effective when their interfaces are well-defined, so this chapter also briefly reviews module design concepts.

Though some coverage here might qualify as advanced and optional, this is mostly a miscellany of additional module subjects. Because some of the topics discussed here are very widely used—especially the `__name__` dual-mode trick—be sure to browse here before moving on to classes in the next part of the book.

Module Design Concepts

First up, some perspective. Like functions, modules present design trade-offs: you have to think about which functions go in which modules, module communication mechanisms, and so on. All of this will become clearer when you start writing bigger Python systems, but here are a few general ideas to keep in mind:

- **You’re always in a module in Python.** There’s no way to write code that doesn’t live in some module. As mentioned briefly in Chapters 17 and 21, even code typed at the interactive prompt (a.k.a. REPL) really goes in a built-in module called `__main__`; the only unique things about the interactive prompt are that code runs and is discarded immediately, and expression results are printed automatically.
- **Minimize module coupling: global variables.** Like functions, modules work best if they’re written to be mostly closed boxes. As a rule of thumb, they should be as independent of global variables used within other modules as possible, except for functions and classes imported from them. The only things a module should share with the outside world are the tools it uses, and the tools it defines.
- **Maximize module cohesion: unified purpose.** Also like functions, you can minimize a module’s couplings by maximizing its cohesion. If all the components of a module share a general purpose, they’re less likely to depend on external names.

- **Modules should rarely change other modules' variables.** We illustrated this with code in [Chapter 17](#), but it's worth repeating here: it's perfectly OK to use globals defined in another module (that's how clients import services, after all), but *changing* globals in another module is often a symptom of a design problem. There are exceptions, of course, but you should try to communicate results through devices such as function arguments and return values, not cross-module changes. Otherwise, your globals' values become dependent on the order of arbitrarily remote assignments in other files, and your modules become harder to understand and reuse.

As a summary, [Figure 25-1](#) sketches the environment in which modules operate. Modules contain variables, functions, and classes, and import other modules for the tools they define. Functions have local variables of their own, as do classes—objects that live within modules and which we'll begin studying in the next chapter. As we saw in [Part IV](#), functions can nest, too, but all are ultimately contained by modules at the top.

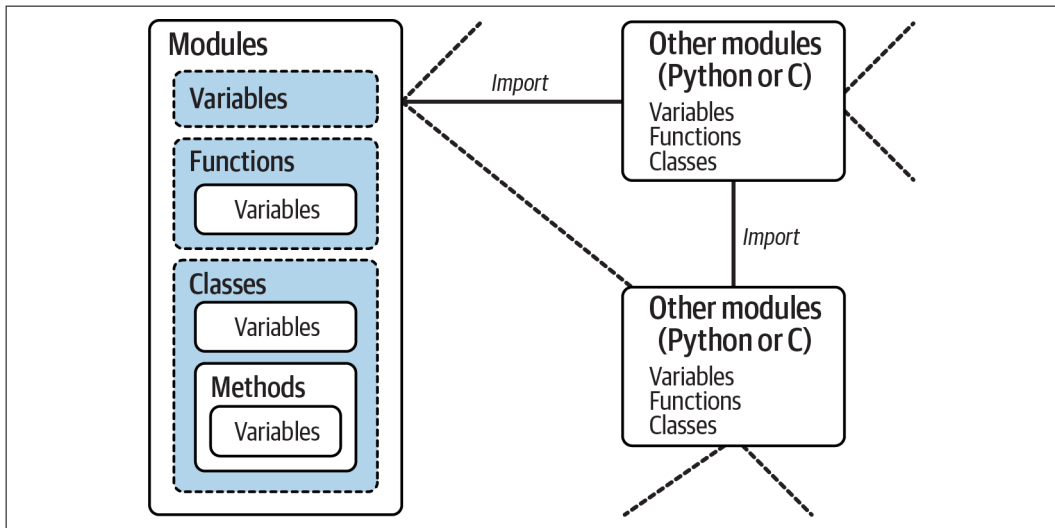


Figure 25-1. Module execution environment

Data Hiding in Modules

Next, we turn to private matters. As we've seen, a Python module exports all the names assigned at the top level of its file. There is no syntax for declaring which names should and shouldn't be visible outside the module. In fact, there's no way to prevent a client from changing names inside a module if it wants to.

In Python, data hiding in modules is a *convention*, not a syntactical constraint. If you want to break a module by trashing its names, you can, but most programmers don't count this as a life goal. Some purists object to this liberal attitude toward data hiding, claiming that it means Python can't implement encapsulation. However, encapsulation in Python is more about packaging than about restricting. We'll expand on this idea in the next part in relation to classes, which also have no privacy syntax but can often emulate its effect in code.

Minimizing from * Damage: `_X` and `__all__`

That being said, as a limited special case, you can prefix names with a single underscore (e.g., `_X`) to prevent them from being copied out when a client imports a module's names with a `from *` statement. This really is intended only to minimize namespace pollution; because `from *` copies out all names, the importer may get more than it's bargained for (including names that overwrite names in the importer). But underscores aren't "private" declarations: you can still see and change such names with other import forms. [Example 25-1](#) demos the idea.

Example 25-1. unders.py

```
a, b, _c, _d = 1, 2, 3, 4          # Control from * exports, take 1
```

When names both with and without underscores are assigned this way, `from *` can't see the former, but `import` and normal `from` can:

```
$ python3
>>> from unders import *          # Load non _X names only on from *
>>> a, b
(1, 2)
>>> _c
NameError: name '_c' is not defined. Did you mean: '_'?

>>> from unders import _c        # But other importers get every name
>>> _c
3
>>> import unders
>>> unders._d
4
```

Alternatively, you can achieve a hiding effect similar to the `_X` naming convention by assigning a list of variable name strings to the variable `__all__` at the top level of the module. When this feature is used, the `from *` statement will copy out *only* those names listed in the `__all__` list, though other imports work as before.

In effect, this is the converse of the `_X` convention: `__all__` identifies names to be *copied*, while `_X` identifies names *not* to be copied. Python looks for an `__all__` list in the module first, and copies its names irrespective of any underscores; if `__all__` is not found, `from *` copies all names without a single leading underscore. To demo, [Example 25-2](#) uses both name-hiding tools.

Example 25-2. alls.py

```
__all__ = ['a', '_c']             # Control from * exports, take 2
a, b, _c, _d = 1, 2, 3, 4        # __all__ has precedence over _X
```

On imports, `from *` gets everything in `__all__`, but no others; other importers again get everything:

```
$ python3
>>> from alls import *                # Load __all__ names - only
>>> a, _c                             # Even if they have underscores
(1, 3)
>>> b
NameError: name 'b' is not defined

>>> from alls import a, b, _c, _d      # But other importers get every name
>>> a, b, _c, _d
(1, 2, 3, 4)

>>> import alls
>>> alls.a, alls.b, alls._c, alls._d
(1, 2, 3, 4)
```

Like the `_X` convention, the `__all__` list has meaning only to the `from *` statement form and does not amount to a privacy declaration: other import statements can still access all names, as the last two tests show. Still, module writers can use either technique to implement modules that are well-behaved when used with `from *`.

See also the discussion of `__all__` lists in package `__init__.py` files in [Chapter 24](#). In this context, these lists declare nested submodules to be automatically loaded for a `from *` run on their container. The effect is similar to name hiding in module files, though packages extend it to apply to the content of a package folder in the filesystem.

Managing Attribute Access: `__getattr__` and `__dir__`

On the subject of data hiding in modules, Python 3.7 added support for special functions at a module's top level that can be used to manage access to a module's attributes. If defined, a module's `__getattr__` function is automatically run when a module attribute is not found, and its `__dir__` overrides the normal attribute-list fetch run for the `dir` built-in. These can be used to implement both basic access constraints and arbitrarily dynamic interfaces.

These functions also shadow same-named tools in *classes* and are meant in part to obviate a long-standing and obscure trick that reset a module's object in the `sys.modules` table to an instance of a *class* with these same methods. This, of course, means that these functions may make more sense after we study classes in [Part VI](#), but the artificial module in [Example 25-3](#) demos the basics.

Example 25-3. gamod.py

```
var = 2                                # Real attribute returned directly

def __getattr__(name):                 # Undefined attr fetches routed here
    print(f'(virtual {name})', end=' ')
    match name:
        case 'test':
            return name * var
        case 'hack' | 'code':
            return name.upper()
        case _:
            raise AttributeError(f'{name} is undefined')

def __dir__():
    return ['var', 'test', 'hack', 'code']
```

When imported, fetches of real attributes defined in the module work normally (subject to the `_X` and `__all__` of the prior section for `from *`), but missing names are routed to `__getattr__`, which can manage the request. It may also use a `raise` statement to flag an invalid request with an *exception*—a topic we'll study in full later in the book because it's also dependent on classes today:

```
>>> import gamod
>>> gamod.var                # Real: __getattr__ not called
2
>>> gamod.test              # Virtual: computed when fetched
(virtual test) 'testtest'
>>> gamod.hack
(virtual hack) 'HACK'
>>> gamod.nonesuch
AttributeError: nonesuch is undefined

>>> dir(gamod)
['code', 'hack', 'test', 'var']
```

The `from` statement invokes `__getattr__` too, though `from *` requires names to be listed on `__all__` (you can largely ignore the spurious `__path__` fetch here, though a `__getattr__` must accommodate it):

```
>>> from gamod import code
(virtual __path__) (virtual code)
>>> code
'CODE'
>>> from gamod import *
(virtual __path__) (virtual __all__)
>>> var
2
>>> hack
NameError: name 'hack' is not defined
```

Importantly, `__getattr__` is *not* run for global-scope lookup within the module itself, so in-file undefined names remain undefined. It's really just for attribute fetches from *other* modules and does not catch *assignments* anywhere. For example, the first line of the following added at the bottom of [Example 25-3](#) would fail, and the second line run in the REPL would make a new attribute in the module which bypasses `__getattr__` thereafter:

```
print(test)                # File: does NOT call __getattr__ (raises NameError)
gamod.hack = 'real'        # REPL: does NOT call __getattr__ (makes attribute)
```

Although this all works as advertised, it is a tool-builder's hook, and you'll have to unearth legitimate use cases. It may be useful in narrow roles, but it also *conflates* modules with classes and discounts the fact that module learners do not already understand these functions' origins in classes. This is a regretably common theme in Python: additions often come with forward dependencies that seem to expect users to already know Python in order to use Python. Python is not just for Python experts, but that's a message baked into many a mod.

The good news here may be that a later proposal to add classes' `__setattr__` for module-attribute assignment was rejected by Python's steering committee—though only after allowing `__getattr__` and `__dir__` to sneak in. As usual, you should weigh the convolutions of this extension against its real-world applications.

Enabling Language Changes: `__future__`

Speaking of changes, Python mods that may break existing code are often introduced gradually. This is not always as “gradual” as it might be, and version 3.0 was a glaring exception (though 2.X was supported for 12 more years after 3.X’s release). Sometimes, though, changes initially appear as optional extensions, which are disabled by default. To enable such an extension in Pythons that predate its official arrival, use a special `import` statement of this form:

```
from __future__ import featurename
```

When coded in a script, this statement must appear as the first executable statement in the file (possibly following a docstring or comment), because it enables special compilation of code on a per-module basis. It’s also possible to submit this statement at the interactive prompt to experiment with upcoming language changes; the feature will then be available for the remainder of the interactive session.

For example, the prior edition of this book used this statement in Python 2.X to activate 3.X true division of [Chapter 5](#), 3.X print calls of [Chapter 11](#), and 3.X absolute imports for packages of [Chapter 24](#). Earlier editions used this statement form to demonstrate generator functions, which require a `yield` that was not yet enabled by default.

This edition is boldly going forward with the present, but `__future__` can be used in older Pythons to enable Python 3.7’s `StopIteration` “bubbling” behavior described at the end of [Chapter 20](#):

```
from __future__ import generator_stop
```

For a list of futurisms you may import and turn on this way, see the Python library manual’s entry for `__future__`. Per its documentation, none of its feature names will ever be removed, so it’s safe to leave in a `__future__` import even in code run by a version of Python where the feature is enabled normally. The future does not erase the past.

Dual-Usage Modes: `__name__` and `__main__`

Our next module-related trick lets you both import a file as a *module* and run it as a standalone *script*, a hook that is widely used in Python files. It’s actually so simple that some learners miss the point at first. Each module has a built-in attribute called `__name__`, which Python creates and assigns automatically as follows:

- If the file is being run as a top-level script file, `__name__` is set to the string `'__main__'` when it starts.
- If the file is being imported instead, `__name__` is set to the module’s name as known by its clients.

The upshot is that a module can test its own `__name__` to determine whether it’s being run or imported. For example, suppose we create the code file named *dualmode.py* in [Example 25-4](#), with a single function called `title`.

Example 25-4. *dualmode.py*

```
def title():
    print('Learning Python, 6E')

if __name__ == '__main__':
    title()                                # Only when run
                                           # Not when imported
```

This module defines a function for clients to import and use as usual:

```
$ python3
>>> import dualmode
>>> dualmode.title()
Learning Python, 6E
```

But the module also includes code at the bottom that is set up to call the function automatically when this file is run as a program:

```
$ python3 dualmode.py
Learning Python, 6E
```

In effect, a module's `__name__` variable serves as a *usage mode flag*, allowing its code to be leveraged as *both* an importable library and a top-level script. Though simple, you'll see this hook used in many of the Python program files you are likely to encounter in the wild—both for testing and dual usage.

For instance, one of the most common ways you'll see the `__name__` test applied is for *self-test* code. In short, you can package code that tests a module's exports in the module itself by wrapping it in a `__name__` test at the bottom of the file. This way, you can use the file in clients by *importing* it, but also test its logic by *running* it from the system shell or other launching scheme.

Coding self-test code at the bottom of a file under the `__name__` test is probably the most common and simplest unit-testing protocol in Python. It's much more convenient than retyping all your tests at the interactive prompt. (Preview: [Chapter 36](#) will discuss other commonly used options for testing Python code—as you'll see, the `unittest` and `doctest` standard-library modules provide more advanced testing tools.)

In addition, the `__name__` trick is also commonly used when you're writing files that can be useful both as command-line utilities and as tool libraries. For instance, suppose you write a file-finder script in Python. You can get more mileage out of your code if you package it in functions, and add a `__name__` test in the file to automatically call those functions when the file is run standalone. That way, the script's code becomes reusable in other programs.



What's in a `__name__`?: Don't confuse the `__main__` hook here with the `__main__.py` file discussed in the prior chapter. That file serves as a script when running an entire package *folder* as a program, but testing whether `__name__` is `'__main__'` is used to give two roles to a single *file*. Python often conflates the same names for similar but different purposes—see `__getattr__`!

Example: Unit Tests with `__name__`

In fact, we've already seen numerous cases in this book where the `__name__` check could be useful. As one example, we coded a script in [Chapter 18](#) that computed the minimum value from the set of arguments sent—[Example 18-3](#), whose code is repeated here for ease of reference:

```

def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))    # Self-test code
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))

```

This script includes self-test code at the bottom, so we can test it without having to retype test code in the REPL each time we run it. The problem with the way it is currently coded, however, is that the output of the self-test call will appear when this file is imported from another file to be used as a tool—not exactly a client-friendly feature! To do better, we can wrap up the self-test call in a `__name__` check so that it will be launched *only* when the file is run as a top-level script, not when it is imported. [Example 25-5](#) lists this new-and-improved version of the module.

Example 25-5. *minmax.py*

```

print('I am:', __name__)

def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

if __name__ == '__main__':
    print(minmax(lessthan, 4, 2, 1, 5, 6, 3))    # Self-test code
    print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))

```

We’re also printing the value of `__name__` at the top here to trace its value (something you wouldn’t do in a real library module). Python creates and assigns this usage-mode variable as soon as it starts loading a file. When we run this file as a top-level script, its name is set to `__main__`, so its self-test code kicks in automatically:

```

$ python3 minmax.py
I am: __main__
1
6

```

If we import the file, though, its name is not `__main__`, so we must explicitly call the function to make it run:

```

$ python3
>>> import minmax
I am: minmax
>>> minmax.minmax(minmax.lessthan, *'hack')
'a'

```

Again, regardless of whether this is used for testing, the net effect is that we get to use our code in *two different roles*—as a library module of tools, or as an executable program. It's buy-one-get-one code. You'll also see programs that route program-mode runs into a module called `main`:

```
def main():
    ...
if __name__ == '__main__':
    main()
```

This works, but it's extra code, and there's nothing special about a function named `main` in Python—unlike some other languages, which may be part of the inspiration for this indirection's appearance in Python code.



Fishing tutorial past: For another example of the `__name__ == '__main__'` test at work, see the dual-mode script/module *formats.py* in this book's examples package. It formats numbers with commas and currency conventions and demos how you can code your own flexible tools instead of relying on built-ins. It didn't add much here and was cut in this edition for space, but provides optional self-study code—and underscores that learning to fish generally beats being given one.

The `as` Extension for `import` and `from`

Next on the tour is a follow-up on a topic introduced in [Chapter 23](#)'s name-collision coverage. As a minor but useful convenience, both the `import` and `from` statements support an optional `as` clause, which simply *renames* a name imported by your script. For example, the following `import` statement using the `as` extension:

```
import modulename as name                # And use name, not modulename
```

is equivalent to the following set of three statements, which renames the module in the importer's scope only (it's still known by its original name to other files), and drops the original name in the importer's scope altogether:

```
import modulename                        # Run a normal import
name = modulename                       # Rename the module - here
del modulename                          # Discard the original name - here
```

After an `import` with `as`, you can—and in fact, must—use the name listed after the `as` to refer to the module. The longer equivalent works because modules are *first-class objects* just like functions, and can be passed around freely.

The `as` extension works in a `from` statement, too, to assign a name imported from a file to a different name in the importer's scope. As before, you get only the new name you provide, not its original:

```
from modulename import attrname as name  # And use name, not attrname
```

This in turn works the same as the following statements:

```
from modulename import attrname
name = attrname
del attrname
```

As noted in [Chapter 23](#), this extension is commonly used both to provide *shorter synonyms* for longer names and to avoid *name clashes* when you are already using a name in your script that would otherwise be overwritten by a normal import:

```

import reallylongmodulename as name          # Use shorter nickname
name.func()                                  # Rename to make shorter

from module1 import utility as util1          # Can have only one "utility"
from module2 import utility as util2          # Rename to make unique
util1(); util2()

```

By way of review, the `as` clause also comes in handy for providing a short, simple name for an entire directory path and avoiding name collisions when using the *package import* feature described in [Chapter 24](#):

```

import dir1.dir2.mod as mod                  # Only list full path once
mod.func()                                  # Only one change if path changes

from dir1.dir2.mod import func as modfunc    # Rename to make unique if needed
modfunc()                                    # Allow func to be something else

```

Finally, the `as` clause is also something of a hedge against name *changes*: if a new release of a library renames a module or tool your code uses extensively, or provides a new alternative you'd rather use instead, you can simply rename it to its prior name on import to avoid breaking your code:

```

import newname as oldname
from library import newname as oldname
...and keep happily using oldname until you have time to update all your code...

```

That said, if all software changes were just name changes, we'd have a lot less to fill our time!

Module Introspection

Next up is module plumbing. Because modules expose most of their interesting properties as built-in attributes, it's easy to write programs that manage other programs—tools we usually call *metaprograms*, because their subjects are other programs. This domain is also referred to as *introspection*, because programs can see and process object internals. Introspection is a somewhat advanced feature, but it can be useful for building programming tools.

For instance, to fetch the value of a module's attribute, we can use attribute qualification or index the module's attribute dictionary, exposed in the built-in `__dict__` attribute we explored in [Chapter 23](#). As we've also seen, Python's `vars` built-in is an alternative way to access `__dict__`, and its `sys.modules` dictionary records all loaded modules by import-name string. In addition, its `getattr` built-in lets us fetch attributes from their string names—it's like saying `object.attr`, but `attr` is an expression that produces a string at runtime.

Hence, all the following expressions reach the same attribute and object named `name` after importing `M` and `sys`:

```

M.name                                     # Qualify object by attribute
M.__dict__['name']                         # Index namespace dictionary manually
vars(M)['name']                           # Namespace dictionary alternative
sys.modules['M'].name                       # Index loaded-modules table manually
getattr(M, 'name')                         # Call built-in fetch function
sys.modules['M'].__dict__['name']           # Module and attribute name strings

```

Demoing with [Example 25-5](#) (and chained comparisons that imply an `and` and a right-side repeat):

```
>>> import minmax, sys
>>> (minmax.lessthan
    is minmax.__dict__['lessthan']    is vars(minmax)['lessthan']
    is sys.modules['minmax'].lessthan is getattr(minmax, 'lessthan')
    is sys.modules['minmax'].__dict__['lessthan'])

True
```

Of course, the first of these is much easier on the eyes (and keyboard), but the others support more generic access.

Example: Listing Modules with `__dict__`

By exposing module internals like this, Python helps you build programs about programs. As a demo, the module in [Example 25-6](#), named *mydir.py*, puts these ideas to work to implement a customized and expanded version of the built-in `dir` function. It defines and exports a function called `listing`, which takes a module object as an argument and prints a formatted display of the module's namespace sorted by attribute name.

Example 25-6. mydir.py

```
"""
mydir.py: a module that lists the namespaces of other modules.
Import this module's listing and pass an imported module, or
run this file as a script to perform its self-test code.
"""

sepchr = '-'
seplen = 60

def listing(module, verbose=True, unders=True):
    """
    List module: just attributes if verbose=False,
    hide built-in __X__ attributes if unders=False.
    """
    sepline = sepchr * seplen
    if verbose:
        print(sepline)
        print(f'name: {module.__name__}\nfile: {module.__file__}')
        print(sepline)

    # Scan namespace keys
    for (count, attr) in enumerate(sorted(module.__dict__)):
        prefix = f'{count + 1:02d} {attr}'
        if attr.startswith('__'):
            if unders:
                print(prefix, '<built-in name>')    # Skip __file__, etc.
            else:
                print(prefix, getattr(module, attr)) # Or module.__dict__[attr]

    if verbose:
        print(sepline)
        print(f'{module.__name__} has {count + 1} names')
        print(sepline)

if __name__ == '__main__':
    import mydir
    listing(mydir)                                # Self-test code: list myself
```

Notice the *docstrings* in this module; because we may want to use this as a general tool, the docstrings provide functional information accessible via `help` and the browser mode of PyDoc—tools that use similar introspection tools to do their jobs (see [Chapter 15](#) for usage info). A *self-test* is also provided at the bottom of this module, which narcissistically imports and lists itself; here’s the sort of output produced (with path edits for space):

```
$ python3 mydir.py
-----
name: mydir
file: /Users/me/.../LP6E/Chapter25/mydir.py
-----
01) __builtins__ <built-in name>
02) __cached__ <built-in name>
03) __doc__ <built-in name>
04) __file__ <built-in name>
05) __loader__ <built-in name>
06) __name__ <built-in name>
07) __package__ <built-in name>
08) __spec__ <built-in name>
09) listing <function listing at 0x1077758a0>
10) sepchr -
11) seplen 60
-----
mydir has 11 names
-----
```

To use this as a tool for listing other modules, simply pass the modules in as objects to this file’s function. Here it is listing itself manually, as well as attributes in the `tkinter` GUI module in the Python standard library; it will technically work on any object with `__name__`, `__file__`, and `__dict__` attributes:

```
>>> from mydir import listing
>>> import mydir, tkinter

>>> listing(mydir, unders=False, verbose=False)
09) listing <function listing at 0x10800fba0>
10) sepchr -
11) seplen 60

>>> listing(tkinter, unders=False)
-----
name: tkinter
file: /.../lib/python3.12/tkinter/__init__.py
-----
01) ACTIVE active
02) ALL all
03) ANCHOR anchor
04) ARC arc
...more names omitted...
166) re <module 're' from '/.../lib/python3.12/re/__init__.py'>
167) sys <module 'sys' (built-in)>
168) types <module 'types' from '/.../lib/python3.12/types.py'>
169) wantobjects 1
-----
tkinter has 169 names
-----
```

You'll meet `getattr` and its relatives again later. The point to notice here is that `mydir` is a program that lets you browse other programs. Because Python exposes its internals, you can process objects generically.



REPL startup tip: You can preload tools such as `mydir.listing` and the reloader we'll code in a moment into the interactive REPL by importing them in a file named by the `PYTHONSTARTUP` environment variable. Because code in the startup file runs in the interactive namespace, importing common tools in this file can save you some typing. See [Appendix A](#) for more info.

Importing Modules by Name String

Finally, it's time for something more dynamic. By now, you've probably noticed that the module name in an `import` or `from` statement is a hardcoded variable name. Sometimes, though, your program will get the name of a module to be imported as a string at runtime—from a user selection in a GUI, or a parse of an XML document, for instance. Unfortunately, you can't use `import` statements directly to load a module given its name as a string—Python expects a variable name that's taken literally and not evaluated, not a string or expression. For instance:

```
>>> import 'string'
SyntaxError: invalid syntax
```

It also won't work to simply assign the string to a variable name:

```
>>> x = 'string'
>>> import x
ModuleNotFoundError: No module named 'x'
```

Here, Python will try to import a file `x.py`, not the `string` module—the name in an `import` statement both becomes a variable assigned to the loaded module and identifies the external file literally.

Running Code Strings

To get around this, you need to use special tools to load a module dynamically from a string that is generated at runtime. The most general approach is to construct an `import` statement as a string of Python code and pass it to the `exec` built-in function to run:

```
>>> modname = 'string'
>>> exec('import ' + modname)      # Run a string of code
>>> string                        # Imported in this namespace
<module 'string' from '/.../lib/python3.12/string.py'>
```

We met the `exec` function—and its cousin for expressions, `eval`—earlier, in Chapters 3, 5, 9, and 10. `exec` compiles a string of code and passes it to the Python interpreter to be executed. In Python, the bytecode compiler is available at runtime, so you can write programs that construct and run other programs like this. By default, `exec` runs the code in the current scope (as if pasted there), but you can get more specific by passing in optional namespace dictionaries. It also has security issues noted earlier in the book, which may be moot in a code string you are building yourself.

Direct Calls: Two Options

The only real drawback to `exec` here is that it must compile the `import` statement each time it runs, and compiling can be slow. Precompiling to bytecode with the `compile` built-in may help for code strings run many times, but in most cases, it's probably simpler and may run quicker to use the built-in `__import__` function to import from a name string. The effect is similar, but `__import__` returns the module object—assign it to a name to keep it:

```
>>> modname = 'string'
>>> string = __import__(modname)
>>> string
<module 'string' from '/.../lib/python3.12/string.py'>
```

Because imports work by invoking `__import__`, it loads the named module normally. The newer standard-library call `importlib.import_module` does the same job; Python's docs describe it as a simplified wrapper around `__import__` for “everyday” use (though our code is growing longer as our tools are growing newer):

```
>>> import importlib
>>> modname = 'string'
>>> string = importlib.import_module(modname)
>>> string
<module 'string' from '/.../lib/python3.12/string.py'>
```

This call works the same as `__import__` in its basic roles, but see Python's manuals for more details on both calls' advanced usage and arguments. Python's docs also seem to prefer the newer `importlib` call for importing by name string, though this seems subjective, either call works, and the imports system has been a frequent morpher.

On callout here: both calls also work for the *package imports* of the prior chapter, but the first returns the *leftmost* component in a package path, and the second returns the *last*—in fact, this is their most prominent difference:

```
>>> import importlib
>>> __import__('email.message')
<module 'email' from '/.../lib/python3.12/email/__init__.py'>
>>> importlib.import_module('email.message')
<module 'email.message' from '/.../lib/python3.12/email/message.py'>
```

The `importlib` call also works for a *package-relative* import string (with leading dots), if also passed the string name of a package path from which to resolve the import. See [Chapter 24](#) for the story of package imports.

Example: Transitive Module Reloads

To tie together and apply some of the topics we've studied, this section develops a module tool that serves as a larger case study to close out this chapter and part. We explored module reloads in [Chapter 23](#), as a way to pick up changes in code without stopping and restarting a program or REPL. When reloading a module, though, Python reloads only that particular module's file; it doesn't automatically reload modules that the file being reloaded happens to import.

For example, if you reload some module A, and A imports modules B and C, the reload applies only to A—not to B and C. The statements inside A that import B and C are rerun during the reload, but they just fetch the already loaded B and C module objects (assuming they've been imported before). In abstract code, here's the file *A.py*:


```
# A.py
import B           # Not reloaded when A is!
import C           # Just imports of already loaded modules: no-ops

$ python3
>>> ...import and use A...
>>> from importlib import reload
>>> reload(A)
```

By default, this means that you cannot depend on reloads to pick up changes in all the modules in your program transitively. Instead, you must use multiple `reload` calls to update the subcomponents independently. This can require substantial work for large systems you're testing interactively. You can design your systems to reload their subcomponents automatically by adding `reload` calls in parent modules like `A`, but this complicates the code.

A recursive reloader

A better approach is to write a general tool to do transitive reloads automatically, by scanning a module's `__dict__` attributes dictionary and checking the type of each attribute's value to find nested modules to reload. Such a utility function could call itself *recursively* to navigate arbitrarily shaped and deep import-dependency chains. The module `__dict__` was introduced in [Chapter 23](#) and employed by *mydir.py* earlier, recursion was explored in [Chapter 19](#), and the `type` call was presented in [Chapter 9](#); we just need to combine these tools for this new role.

To this end, the module *reloadall.py* listed in [Example 25-7](#) defines a `reload_all` function that automatically reloads a module, every module that the module imports, and so on, all the way to the bottom of each import chain. It uses a dictionary to keep track of already reloaded modules, recursion to walk the import chains, and the standard library's `types` module, which simply predefines `type` results for built-in types like modules. Its `visited` dictionary avoids repeats when imports are recursive or redundant (module objects are immutable, and so can be dictionary keys); as we saw in [Chapters 5](#) and [8](#), a *set* could work similarly (and will in rewrites ahead).

Example 25-7. *reloadall.py*

```
"""
reloadall.py: transitively reload nested modules.
Call reload_all with one or more imported modules as arguments.
These modules, and all the modules they import, are reloaded.
"""

import types
from importlib import reload

def status(module):
    print('reloading', module.__name__)

def tryreload(module):
    try:
        reload(module)
    except:
        print('FAILED:', module)
        # Imports might fail

def transitive_reload(module, visited):
    if not module in visited:
        status(module)
        tryreload(module)
        visited[module] = True
        # Trap cycles, duplicates
        # Reload this module
        # And visit children
```

```

        for attrobj in module.__dict__.values():
            if type(attrobj) == types.ModuleType:
                transitive_reload(attrobj, visited)

def reload_all(*args):
    visited = {}
    for arg in args:
        if type(arg) == types.ModuleType:
            transitive_reload(arg, visited)

def tester(reloader, modname):
    import importlib, sys
    if len(sys.argv) > 1:
        modname = sys.argv[1]
    module = importlib.import_module(modname)
    reloader(module)

if __name__ == '__main__':
    tester(reload_all, 'reloadall')

```

For all attrs in mod
Recur if nested module

Main entry point
For all passed in

Self-test: cmd or passed
Imports for tests only
Command-line argument?

Import by name string
Test passed-in reloader

Test: reload self or arg

Besides namespace dictionaries, this script makes use of other tools we’ve studied before: it includes a `__name__` test to launch self-test code when run as a top-level script only, and its `tester` function uses `sys.argv` to inspect command-line arguments and `importlib` to import a module by name string passed in as a function or command-line argument. Review earlier coverage for more info if needed.

One curious bit: notice how this code’s `tryreload` wraps the basic `reload` call in a `try` statement to catch exceptions. Reloads may fail for many reasons, and it’s best to be defensive when using system interfaces. As you’ll see in a moment, for example, an unreloadable monitoring module added to `sys` in Python 3.12 would otherwise crash the reloader. The `try` was previewed in [Chapter 10](#) and will be covered in full in [Part VII](#).

Testing recursive reloads

To use this module normally, import its `reload_all` function and pass it an already loaded module object—just as you would for the built-in `reload` function. Like `reload`, its module argument is usually obtained by a top-level `import`; as we’ve seen, `sys.modules` fetches work too, but modules accessed only by `from` don’t apply.

To test first, run the module *standalone*. Its `tester` function runs a passed-in reloader on a module imported by *name string*—which is taken from a command-line argument if present, else a passed-in name. In this mode, the module’s self-test code calls `tester` to run `reload_all` on its own imported module by default if no command-line arguments are used (its own name is not defined in the file without an `import`):

```

$ python3 reloadall.py
reloading reloadall
reloading types

```

With a command-line *argument*, the tester instead reloads the listed module by its name string—in the following, the [Chapter 21](#) folder’s benchmark module we coded in [Example 21-8](#). To run this live, you need both the reloader module here and the module it reloads. One way to handle this is to add [Chapter 21](#)’s code folder to your `PYTHONPATH` setting per [Chapter 22](#). Copying files in either direction is subpar, and simply running in the *Chapter21* folder won’t work because the reloader script’s *Chapter25* folder is “home.” Note that we give a *module* name in this mode, not a filename; because the script imports the module using the search path just like `import`, the `.py` extension is omitted:

```

$ pwd                                # In Chapter 25's code folder
/Users/me/.../LP6E/Chapter25
$ export PYTHONPATH=../Chapter21     # Extend path: your shell may vary
$ python3 reloadall.py pybench       # Import+reload Chapter 21 module
reloading pybench
reloading sys
reloading sys.monitoring
FAILED: <module 'sys.monitoring'>
reloading os
reloading abc
reloading stat
reloading posixpath
reloading genericpath
reloading time
reloading timeit
reloading gc
reloading itertools

```

More usefully, we can also deploy this module at the *interactive* prompt. This works like the built-in `reload`, but adds recursive reloads for the module or modules passed—here, for standard-library modules:

```

$ python3
>>> from reloadall import reload_all    # Reload stdlib modules in REPL mode
>>> import os, tkinter
>>> reload_all(os)
reloading os
reloading abc
reloading sys
reloading sys.monitoring
FAILED: <module 'sys.monitoring'>
reloading stat
reloading posixpath
reloading genericpath

>>> reload_all(tkinter)
reloading tkinter
reloading collections
reloading collections.abc
...etc...
reloading _sre
reloading functools
reloading copyreg

```

In either mode, the reloader also works on module *packages*—here, for the standard library’s `email` package:

```

$ python3 reloadall.py email.message # Import+reload a stdlib package
reloading email.message
reloading binascii
reloading re
...etc...

$ python3
>>> from reloadall import reload_all    # Same, but in REPL mode
>>> import email.message
>>> reload_all(email.message)
reloading email.message
reloading binascii
reloading re
...etc...

```

The following runs the reloader on the `dir1.dir2.mod` path we coded in “Basic Package Structure” on page 595. All items in the path are reloaded from a package root, and a `sys.path` mod in the REPL gives import access to another chapter’s code folder—much like the `PYTHONPATH` setting used earlier (again, per Chapter 22):

```
>>> import sys
>>> sys.path.append('../Chapter24')           # Extend the search path manually
>>> import dir1.dir2.mod                     # A package in Chapter 24's folder
Running dir1.__init__.py
Running dir1.dir2.__init__.py
Loading dir1.dir2.mod

>>> reload_all(dir1)                         # Reloads all on path: mods in mods
reloading dir1
Running dir1.__init__.py
reloading dir1.dir2
Running dir1.dir2.__init__.py
reloading dir1.dir2.mod
Loading dir1.dir2.mod
```

Finally, here is a simple session that demos the effect of normal versus *transitive* reloads—changes made to the two nested files are not picked up by reloads unless our transitive utility is used (files are listed inline here for brevity):

```
# File ra.py
import rb
X = 1

# File rb.py
import rc
Y = 2

# File rc.py
Z = 3

$ python3
>>> import ra
>>> ra.X, ra.rb.Y, ra.rb.rc.Z               # Three-level import chain
(1, 2, 3)
```

Now, without stopping Python, change all three files’ assignment values, save the files, and reload back in the REPL:

```
>>> from importlib import reload
>>> reload(ra)                               # Built-in reload is top-level only
<module 'ra' from '/.../LP6E/Chapter25/ra.py>
>>> ra.X, ra.rb.Y, ra.rb.rc.Z
(111, 2, 3)

>>> from reloadall import reload_all
>>> reload_all(ra)                           # Normal usage mode
reloading ra
reloading rb
reloading rc
>>> ra.X, ra.rb.Y, ra.rb.rc.Z               # Reloads all nested modules too
(111, 222, 333)
```

This is similar to the preceding package-path reload, but the imports here are explicit; module nesting in packages is implied. Study the reloader's code and results for more on its operation. The next section exercises its tools further.

Alternative codings

For all the recursion fans in the audience (and we know who we are), [Example 25-8](#) lists an alternative *recursive* coding for the original function in [Example 25-7](#). This new version uses a *set* instead of a dictionary to detect repeats and cycles, is marginally more *direct* because it eliminates a top-level loop, and serves to illustrate recursive coding in general. Compare with the original to see how this differs.

This version also gets some of its work for free from the original; in fact, this module essentially *extends* the original to replace just the parts that vary. Notice how it calls the original version's tester, passing in the `reload_all` defined here—which ensures that this module's reloader is run when this script is launched in standalone mode.

Example 25-8. reloadall2.py

```
"""
reloadall2.py: transitively reload nested modules.
Alternative coding: recursive, refactored.
"""

import types
from reloadall import status, tryreload, tester

def transitive_reload(objects, visited):
    for obj in objects:
        if type(obj) == types.ModuleType and obj not in visited:
            status(obj)
            tryreload(obj)                # Reload this, recur to attrs
            visited.add(obj)
            transitive_reload(obj.__dict__.values(), visited)

def reload_all(*args):
    transitive_reload(args, set())

if __name__ == '__main__':
    tester(reload_all, 'reloadall2')    # Test: reload myself or arg
```

As we saw in [Chapter 19](#), there is usually an *explicit stack* or *queue* equivalent to recursive functions, which may be preferable in some contexts. [Example 25-9](#) lists one such transitive reloader—it uses a stack instead of recursion, and a *set* to skip repeats and cycles. On each loop, all of a new module's attribute values are added to the end of the objects stack and filtered later, and this is repeated until the list of candidate objects becomes empty. A generator expression could filter out nonmodules in the `extend` call to avoid some pops, but this would be more complex.

Because it both pops and adds items at the *end* of its list, this version is stack-based, though the order of both pushes and dictionary values influences the order in which it reaches and reloads modules—it visits submodules in namespace dictionaries from *right to left*, unlike the left-to-right order of the recursive versions (trace through the code to see how). We could change this to match by reversing values, but reload order is unimportant.

Example 25-9. reloadall3.py

```
"""
reloadall3.py: transitively reload nested modules.
Alternative coding: nonrecursive, explicit stack.
"""

import types
from reloadall import status, tryreload, tester

def transitive_reload(objects, visited):
    while objects:
        next = objects.pop()                # Delete next item at end
        if (type(next) == types.ModuleType  # Is it a module object?
            and next not in visited):        # Not already reloaded?
            status(next)                    # Reload this, push attrs
            tryreload(next)
            visited.add(next)
            objects.extend(next.__dict__.values())

def reload_all(*args):
    transitive_reload(list(args), set())

if __name__ == '__main__':
    tester(reload_all, 'reloadall3')        # Test: reload myself or arg
```

If the recursion and nonrecursion used in these examples is confusing, see the discussion of recursive functions in [Chapter 19](#) for more background on the subject.

Testing reload variants

To prove that these two alternative reloaders work the same as the original, let's test all three of our reloader variants. Thanks to their common testing function, we can run all three from a command line both with no arguments to test the module reloading itself, and with the name of a module to be reloaded listed on the command line (in `sys.argv`):

```
$ python3 reloadall.py
reloading reloadall
reloading types

$ python3 reloadall2.py
reloading reloadall2
reloading types

$ python3 reloadall3.py
reloading reloadall3
reloading types
```

Though it's hard to see here, we really are testing the individual reloader alternatives—each of these tests shares a common tester function but passes it the `reload_all` from its own file. Here are the variants reloading the `tkinter` GUI module and all the modules its imports reach; again, the third's reloads order varies:

```
$ python3 reloadall.py tkinter
reloading tkinter
reloading collections
reloading collections.abc
...etc...
$ python3 reloadall2.py tkinter
```

```

reloading tkinter
reloading collections
reloading collections.abc
...etc...
$ python3 reloadall3.py tkinter
reloading tkinter
reloading re
reloading copyreg
...etc...

```

As usual, we can test interactively, too, by importing and calling either a module's main reload entry point with a module object, or the testing function with a reloader function and module name string:

```

$ python3
>>> import reloadall, reloadall2, reloadall3
>>> import tkinter
>>> reloadall.reload_all(tkinter)                                # Normal use case
reloading tkinter
reloading collections
reloading collections.abc
...etc...
>>> reloadall.tester(reloadall2.reload_all, 'tkinter')          # Testing utility
reloading tkinter
reloading collections
reloading collections.abc
...etc...
>>> reloadall.tester(reloadall3.reload_all, 'reloadall3')       # Mimic self-test code
reloading reloadall3
reloading types

```

Finally, as noted, the third reloader's results will generally vary by *order*; reload order in all reloaders depends on namespace dictionary ordering (which, as we've learned is deterministically ordered by key insertion time today), but the last also relies on the order in which items are added to its stack. To ensure that all three are reloading the same modules irrespective of the order in which they do so, we can use *sets* or *sorts* to test for order-neutral equality of their printed messages—obtained here by running shell commands with the `os.popen` utility we used in [Chapter 21](#):

```

>>> import os
>>> res1 = os.popen('python3 reloadall.py tkinter').readlines()
>>> res2 = os.popen('python3 reloadall2.py tkinter').readlines()
>>> res3 = os.popen('python3 reloadall3.py tkinter').readlines()

>>> res1[:3]
['reloading tkinter\n', 'reloading collections\n', 'reloading collections.abc\n']
>>> res3[:3]
['reloading tkinter\n', 'reloading re\n', 'reloading copyreg\n']

>>> res1 == res2, res2 == res3
(True, False)
>>> set(res2) == set(res3)                                # Order-neutral equality
True
>>> sorted(res2) == sorted(res3)                            # Ditto
True

```

Run these scripts, study their code, and experiment on your own for more insight; these are the sort of importable tools you might want to add to your own source code library. Watch for a similar testing technique in the coverage of class tree listers in [Chapter 31](#), where we'll apply it to passed *class* objects and extend it further.

Caveats: keep in mind that all the transitive reloaders, like the `reload` built-in that they use, rely on the fact that module reloads update module objects *in place*, such that all references to those modules in any namespace will see the updated version automatically. Because `from` importers copy names out, they are not updated by reloads, transitive or not. Perhaps worse, modules imported only by `from` won't be reloaded, because they do not exist in any importer's namespace scanned. Doing better may require either source code analysis or import customization.

Tool impacts like this are perhaps another reason to prefer `import` to `from`—which brings us to the end of this chapter and part, and the standard set of warnings for this part's topic.

Module Gotchas

In this section, we'll explore the usual collection of boundary cases that can make life interesting for Python beginners. Some are review here, and a few are so obscure that coming up with representative examples can be a challenge, but most illustrate something important about the language.

Module Name Clashes: Package and Package-Relative Imports

If you have two modules of the same name, you may only be able to import one of them—by default, the one whose directory is leftmost in the `sys.path` module search path will always be chosen. This isn't an issue if the module you prefer is in your top-level script's directory; since that is always first in the module path, its contents will be located first automatically. For cross-directory imports, however, the linear nature of the module search path means that same-named files can clash.

To fix this, either avoid same-named files or use the package imports feature of [Chapter 24](#). If you really need to get to two files of the same name, the latter is the solution: structure your source files in subdirectories, such that package-import directory names make the module references unique. As long as the enclosing package directory names are unique, you'll be able to access either or both of the same-named modules.

This issue can also crop up if you accidentally use a name for a module of your own that happens to be the same as a standard-library module you need—your local module in the program's home directory (or another directory early in the module path) can hide and replace the library module.

To fix that, either avoid using the same name as another module you need or store your modules in a package directory and use the package-relative import model of [Chapter 24](#). In this model, normal imports skip the package directory to access the library's version, but special dotted import statements can still select the local version of the module.

Statement Order Matters in Top-Level Code

As we've seen, when a module is first imported (or later reloaded), Python executes its statements one by one, from the top of the file to the bottom. This has a few subtle implications regarding *forward references* that are worth underscoring here:

- Code at the *top level* of a module file (not nested in a function) runs as soon as Python reaches it during an import; because of that, it cannot reference names assigned *lower* in the file.
- Code inside a *function* body doesn't run until the function is called; because names in a function aren't resolved until the function actually runs, they can usually reference names *anywhere* in the file.

In other words, forward references are usually only a concern in top-level module code that executes immediately; functions can reference names arbitrarily. Here's a file that illustrates forward reference dos and don'ts:

```
func1()                                # Error: func1 not yet assigned

def func1():
    print(func2())                     # OK: func2 looked up later

func1()                                # Error: func2 not yet assigned

def func2():
    return "Hello"

func1()                                # OK: func1 and func2 assigned
```

When this file is imported (or run as a standalone program), Python executes its statements from top to bottom. The first call to `func1` fails because the `func1` `def` hasn't run yet. The call to `func2` inside `func1` works as long as `func2`'s `def` has been reached by the time `func1` is called—and it hasn't when the second top-level `func1` call is run. The last call to `func1` at the bottom of the file works because `func1` and `func2` have both been assigned.

Mixing `defs` with top-level code is not only difficult to read, but it's also dependent on statement ordering. As a rule of thumb, if you need to mix immediate code with `defs`, put your `defs` at the top of the file and your top-level code at the bottom. That way, your functions are guaranteed to be defined and assigned by the time Python runs the code that uses them.

from Copies Names but Doesn't Link

Although it's commonly used, the `from` statement is the source of a variety of potential gotchas in Python. As we've seen, the `from` statement is really an assignment to names in the importer's scope—a *name-copy* operation, not a name aliasing. The implications of this are the same as for all assignments in Python, but they're especially subtle for names that live in different files. As a refresher, suppose we define the simple module *nested.py* in [Example 25-10](#).

Example 25-10. nested.py

```
X = 99
def printer(): print(X)
```

If we import its two names using `from` in another module (or the REPL, which stands in for one), we get copies of those names, not links to them. Changing a name in the importer resets only the binding of the local version of that name, not the name in *nested1.py*:

```
>>> from nested import X, printer      # Copy names out
>>> X = 88                             # Changes my X only!
>>> printer()                          # nested1's X is still 99
99
```

If we instead use `import` to get the whole module and assign to a qualified name, we change the name in *nested1* (the module's loaded image, not its source code). Attribute qualification directs Python to a name in the module object, rather than a name in the importer:

```
>>> import nested                # Get module as a whole
>>> nested.X = 88                # Change nested1's X
>>> nested.printer()
88
```

Takeaway: changes to names obtained with `from` don't impact any other module. As covered in [Chapter 23](#), changes to mutable *objects* shared by names copied with `from` can impact other modules, but name changes cannot.

from * Can Obscure the Meaning of Variables

This was mentioned earlier but its demo was saved for here. Because you don't list the variables you want when using the `from *` statement form, it can accidentally overwrite names you're already using in your scope. Worse, it can make it difficult to determine where a variable comes from. This is especially true if the `from *` form is used on more than one imported file.

For example, if you use `from *` on three modules in the following, you'll have no way of knowing what a raw function call really means, short of searching all three external module files—all of which may be in other directories:

```
>>> from module1 import *        # May overwrite names silently
>>> from module2 import *        # No way to tell what we get
>>> from module3 import *        # No way to see name origins

>>> func()                       # Huh?
```

The solution is simply not to do this: list the attributes you want in most `from` statements, and use at most one `from *` per file. That way, any undefined names must by deduction be in the module named in the single `from *`. You can avoid the issue altogether if you always use `import` instead of `from`, but that advice is too harsh; like much else in programming, `from` is a convenient tool if used wisely. Even this example isn't an absolute evil—it's OK for a program to use this technique to collect names in a single module for convenience, as long as it's well known.

reload May Not Impact from Imports

Here's another `from`-related gotcha: as discussed previously, because `from` copies (assigns) names when run, there's no link back to the modules where the names came from. Names imported with `from` simply become references to objects, which happen to have been referenced by the same names in the importee when the `from` ran.

Because of this behavior, reloading the module of origin has no effect on clients that import its names using `from`. That is, the client's names will still reference the *original* objects fetched with `from`, even if the names in the original module are later reset. Here's the story in abstract code:

```
from module import X              # X may not reflect any module reloads
...
from importlib import reload
reload(module)                   # Changes module, but not my names
X                                # Still references old object!
```

To make reloads more effective, use `import` and name qualification instead of `from`. Because qualifications always go back to the module, they will find the new bindings of module names after reloading has updated the module's content in place:

```
import module                                # Get module, not names
...
from importlib import reload
reload(module)                               # Changes module in place
module.X                                     # Get current X: reflects module reloads
```

This is why our transitive reloader earlier in this chapter doesn't apply to names fetched with `from`, only `import`; again, if you're going to use reloads, you're probably better off with `import`.

reload, from, and Interactive Testing

In fact, the prior gotcha is even more nuanced than it appears. [Chapter 3](#) warned that it's usually better not to launch programs with imports and reloads because of the complexities involved. Things get even worse when `from` is brought into the mix. Python beginners most often stumble onto its issues in scenarios like this—imagine that after opening a module file in a text edit window, you launch an interactive session to load and test your module with `from`:

```
from module import function
function(1, 2, 3)
```

Finding a bug, you jump back to the edit window, make a change, and try to reload the module this way:

```
from importlib import reload
reload(module)
```

This doesn't work, because the `from` statement assigned only the name `function`, not `module`. To refer to the module in a `reload`, you have to first bind its name with an `import` statement at least once:

```
from importlib import reload
import module
reload(module)
function(1, 2, 3)
```

But this doesn't quite work either—`reload` updates the module object in place, but as discussed in the preceding section, names like `function` that were copied out of the module in the past still refer to the *old objects*; in this instance, `function` is still the original version of the function. To really get the new function, you must refer to it as `module.function` after the `reload`, or rerun the `from`:

```
from importlib import reload
import module
reload(module)
from module import function           # Or give up and use module.function()!
function(1, 2, 3)
```

Now, the new version of the function will finally run, but it seems an awful lot of work to get there.

As you can see, there are problems inherent in using `reload` with `from`: not only do you have to remember to reload after imports, but you also have to remember to rerun your `from` statements after reloads. This is complex enough to trip up even an expert once in a while. In fact, the situation grew even worse with Python 3.X, because you must also remember to import `reload` itself!

The short story is that you should not expect `reload` and `from` to play together nicely. Again, the best policy is not to combine them at all—use `reload` with `import`, or launch your programs other ways, as suggested in [Chapter 3](#): using menu options in IDLE, file icon clicks, system command lines, the `exec` built-in function, or other.

Recursive from Imports May Not Work

The most bizarre (and, thankfully, obscure) gotcha has been saved for last. Because imports execute a file's statements from top to bottom, you need to be careful when using modules that import each other. This is often called *recursive* imports, but the recursion doesn't really occur (in fact, *circular* may be a better term here)—such imports won't get stuck in infinite importing loops. Still, because the statements in a module may not all have been run when it imports another module, some of its names may not yet exist.

If you use `import` to fetch the module as a whole, this probably doesn't matter; the module's names won't be accessed until you later use qualification to fetch their values, and by that time the module is likely complete. But if you use `from` to fetch specific names, you must bear in mind that you will only have access to names in that module that have already been assigned when a recursive import is kicked off.

As a demo of this phenomenon, consider the modules `recur1` and `recur2`, in Examples 25-11 and 25-12.

Example 25-11. `recur1.py`

```
X = 1
import recur2          # Run recur2 now if it doesn't exist
Y = 2
```

Example 25-12. `recur2.py`

```
from recur1 import X    # OK: X already assigned
from recur1 import Y    # Error: Y not yet assigned
```

Module `recur1` assigns a name `X` and then imports `recur2` before assigning the name `Y`. At this point, `recur2` can fetch `recur1` as a whole with an `import`—it already exists in Python's internal modules table, which makes it importable, and also prevents the imports from looping. But if `recur2` uses `from`, it will be able to see only the name `X`; the name `Y`, which is assigned below the `import` in `recur1`, doesn't yet exist, so you get an error:

```
$ python3
>>> import recur1
ImportError: cannot import name 'Y' from partially initialized module 'recur1'
(most likely due to a circular import) (../LP6E/Chapter25/recur1.py)
```

Python avoids rerunning `recur1`'s statements when they are imported recursively from `recur2` (otherwise the imports would send the script into an infinite loop that might require a Ctrl+C solution or worse), but `recur1`'s namespace is incomplete when it's imported by `recur2`.

The solution? Don't use `from` in recursive imports (no, really!). Python won't get stuck in a cycle if you do, but your programs will once again be dependent on the order of the statements in the modules. In fact, there are two ways out of this gotcha:

- You can usually eliminate import cycles like this by careful design—maximizing cohesion and minimizing coupling per the start of this chapter are good first steps.

- If you can't break the cycles completely, postpone module name accesses by using `import` and attribute qualification (instead of `from` and direct names), or by running your `from`s either inside functions (instead of at the top level of the module) or near the bottom of your file to defer their execution.

There is additional perspective on this issue in the exercises at the end of this chapter—which we've officially reached.

Chapter Summary

This chapter surveyed module topics, some of which qualify as advanced. We studied data-hiding techniques, enabling new language features with the `__future__` module, the `__name__` usage-mode variable, transitive reloads, importing by name strings, and more. We also explored module design issues, wrote some substantial programs, and looked at common mistakes related to modules to help you avoid them in your code.

The next chapter begins our exploration of Python's *class*—its object-oriented programming tool. Much of what we've covered in the last few chapters will apply there, too: classes live in modules and are namespaces as well, but they add an extra component to attribute lookup called *inheritance search*. As this is the last chapter in this part of the book, however, before we dive into classes, be sure to work through this part's set of lab exercises. And before that, here is this chapter's quiz to review the topics covered here.

Test Your Knowledge: Quiz

1. What is significant about variables at the top level of a module whose names begin with a single underscore?
2. What does it mean when a module's `__name__` variable is the string `'__main__'`?
3. How might you step through all the attributes in a module with a loop?
4. If the user interactively types the name of a module to test, how can your code import it?
5. If the module `__future__` allows us to import from the future, can we also import from the past?

Test Your Knowledge: Answers

1. Variables at the top level of a module whose names begin with a single underscore are *not* copied out to the importing scope when the `from *` statement form is used. They can still be accessed by an `import` or the normal `from` statement form, though. The `__all__` list is similar, but the logical converse; its contents are the only names that *are* copied out for a `from *`.
2. If a module's `__name__` variable is the string `'__main__'`, it means that the file is being executed as a top-level script instead of being imported from another file in the program. That is, the file is being used as a program, not a library. This usage mode variable supports dual-mode code and tests.
3. By using the module's built-in `__dict__` attribute. This is a normal dictionary that holds all of the module's attributes, so code can iterate over its keys, values, or key/value pairs. When needed, attribute values can also be fetched for string names by indexing `__dict__`, or calling the `getattr` built-in function.

4. User input usually comes into a script as a string; to import the referenced module given its string name, you can build and run an `import` statement with `exec`, or pass the string name in a call to the `__import__` or `importlib.import_module` functions.
5. No, we can't import from the past in Python. We can install (or stubbornly use) an older version of the language, but the latest Python is generally the best Python (with apologies to 2.X fans in the audience).

Test Your Knowledge: Part V Exercises

See “Part V, Modules and Packages” on page 1177 in Appendix B for the solutions.

1. *Import basics*: Write a program that counts the lines and characters in a file (similar in spirit to part of what `wc` does on Unix). With your text editor, code a Python module called *mymod.py* that exports three top-level names:
 - A `countLines(name)` function that reads an input file specified by string *name*, and counts the number of lines in it (hint: `file.readlines` does most of the work for you, and `len` does the rest, though you could count with `for` and file iterators to support massive files too).
 - A `countChars(name)` function that reads an input file and counts the number of characters in it (hint: `file.read` returns a single string, which may be used in similar ways).
 - A `test(name)` function that calls both counting functions with a given input filename. Such a filename generally might be passed in, hardcoded, input from a user like you with the `input` built-in function, or pulled from a command line via the `sys.argv` list demoed in this chapter's *reloadall.py* example; for now, you can assume it's a passed-in function argument.

All three *mymod* functions should expect a filename string to be passed in. If you type more than two or three lines per function, you're working much too hard—use the hints given!

Next, test your module interactively, using `import` and attribute references to fetch your exports. Does your `PYTHONPATH` need to include the directory where you created *mymod.py*? Try running your module on itself: for example, `test('mymod.py')`. Note that `test` opens the file twice; if you're feeling ambitious, you may be able to improve this by passing an open file object into the two count functions (hint: `file.seek(0)` is a file rewind).

2. *from/from **: Test your *mymod* module from exercise 1 interactively by using `from` to load the exports directly, first by name, then using the `from *` variant to fetch everything.
3. *__main__*: Add a line in your *mymod* module that calls the `test` function automatically only when the module is run as a script, not when it is imported. The line you add will probably test the value of `__name__` for the string `'__main__'`, as shown in this chapter. Try running your module from the system command line or other program-launch scheme; then, import the module and test its functions interactively. Does it still work in both modes?
4. *Nested imports*: Write a second module, *myclient.py*, that imports *mymod* and tests its functions; then run *myclient* from the system command line or other scheme. If *myclient* uses `from` to fetch from *mymod*, will *mymod*'s functions be accessible from the top level of *myclient*? What if it imports with `import` instead? Try coding both variations in *myclient* and test interactively by importing *myclient* and inspecting its `__dict__` attribute.

5. *Package imports*: Import your *mymod.py* file from a package. Create a subdirectory called *mypkg* nested in a directory on your module import search path, copy or move the *mymod.py* module file you created in exercise 1 or 3 into the new directory, and try to import it with a package import of the form `import mypkg.mymod` and call its functions. Try to fetch your counter functions with a `from` too.

This works on all Python platforms (that's part of the reason Python uses `"` as a path separator). The package directory you create can be simply a subdirectory of the one you're working in; if it is, it will be found via the home directory component of the search path, and you won't have to configure your path.

You also don't need an `__init__.py` file in the package directory your module was moved into to make this go, but make one with some basic prints in it and see if they run on each import or reload of the package folder. Finally, also copy *mymod.py* to the package folder's `__main__.py` and invoke it by running the folder itself; does it make sense to do that here? Can you still run the nested *mymod.py* module itself?

6. *Reloads*: Experiment with module reloads: if you haven't already, perform the tests in [Chapter 23's *changer.py* \(Example 23-10\)](#), changing the called function's message or behavior repeatedly, without stopping the Python REPL. Depending on your device, you might edit *changer* in another window, or suspend the Python interpreter and edit in the same window (on Unix, a `Ctrl+Z` key combination usually suspends the current process, and an `fg` command later resumes it, though a separate text-editor window can work just as well).
7. *Circular imports*: In the section on recursive (a.k.a. circular) import gotchas, importing *recur1* raised an error. But if you restart Python and import *recur2* interactively, the error doesn't occur—test this and see for yourself. Why do you think it works to import *recur2*, but not *recur1*? (Hint: Python records new modules before running their code, and later imports fetch the module first, whether the module is “complete” yet or not.)

Now, run *recur1* as a top-level script file: `python3 recur1.py`. Do you get the same error that occurs when *recur1* is imported interactively? Why? (Hint: when modules are run as programs, they aren't imported, so this case has the same effect as importing *recur2* interactively; *recur2* is the first module imported.) What happens when you run *recur2* as a script? Circular imports are uncommon in practice. On the other hand, if you can understand why they are a potential problem, you know a lot about Python's import semantics.

Classes and OOP

OOP: The Big Picture

So far in this book, we've been using the term "object" generically. Really, the code written up to this point has been *object-based*—we've passed objects around our scripts, used them in expressions, called their methods, and so on. For our code to qualify as being truly *object-oriented* (OO), though, our objects will generally need to also participate in something called an *inheritance hierarchy*.

This chapter begins our exploration of the Python *class*—a coding structure and device used to implement new kinds of objects in Python that support inheritance. Classes are Python's main object-oriented programming (OOP) tool, so we'll also study OOP basics along the way in this part of the book. OOP offers a different and often more effective way of programming. Like functions, we can use classes to factor code to minimize *redundancy*. Unlike functions, classes make it easy to write new programs by *customizing* existing code instead of changing it in place.

In Python, classes are created with a new statement: the `class`. As you'll see, the objects defined with classes can look a lot like the built-in object types we employed earlier in the book. In fact, classes really just apply and extend the ideas we've already covered; roughly, they are packages of functions that use and process built-in objects. Classes, though, are designed to create and manage new objects, and support *inheritance*—a mechanism of code customization and reuse above and beyond anything we've seen so far.

One note up front: in Python, OOP is entirely optional, and you don't need to use classes just to get started. You can get plenty of work done with simpler constructs such as functions, or even simple top-level script code. Because using classes well requires some up-front planning, they tend to be of more interest to people who work in *strategic* mode (doing long-term product development) than to people who work in *tactical* mode (where time is in very short supply).

Still, as you'll see in this part of the book, classes turn out to be one of the most useful tools Python provides. When used well, classes can actually cut development time radically. They're also employed in popular Python libraries, so most Python programmers will usually find at least a working knowledge of class basics helpful.

Why Use Classes?

Remember when this book told you that programs "do things with stuff" in Chapters 4 and 10? In simple terms, classes are just a way to define new sorts of *stuff*, reflecting real objects in a program's domain. For instance, suppose we decide to implement that hypothetical pizza-making robot we used

as an example in [Chapter 16](#). If we implement it using classes, we can model more of its real-world structure and relationships. Two aspects of OOP could be useful here:

Inheritance

Pizza-making robots are kinds of robots, so they possess the usual robot-y properties. In OOP terms, we say they “inherit” properties from the general category of all robots. These common properties need to be implemented only once for the general case and can be reused in part or in full by all types of robots we may build in the future.

Composition

Pizza-making robots are really collections of components that work together as a team. For instance, for our robot to be successful, it might need arms to roll dough, motors to maneuver to the oven, and so on. In OOP parlance, our robot is an example of composition; it contains other objects that it activates to do its bidding. Each component might be coded as a class, which defines its own behavior and relationships.

While you may never build pizza-making robots, general OOP ideas like inheritance and composition apply to any application that can be decomposed into a set of objects. For example, in typical *GUI* systems, interfaces are written as collections of widgets—buttons, labels, and so on—which are all drawn when their container is drawn (*composition*). Moreover, we may be able to write our own custom widgets—buttons with unique fonts, labels with new color schemes, and the like—which are specialized versions of more general interface devices (*inheritance*).

From a more concrete programming perspective, classes are Python *program units*, just like functions and modules: they are another compartment for packaging logic and data. In fact, classes also define new namespaces, much like modules. But, compared to other program units we’ve already seen, classes have three critical distinctions that make them more useful when it comes to building new objects:

Multiple instances

Classes are essentially *factories* for generating one or more objects. Every time we call a class, we generate a new object with a distinct namespace. Each object generated from a class has access to the class’s attributes *and* gets a namespace of its own for data that varies per object. This is similar to the per-call state retention of [Chapter 17](#)’s *closure* functions, but is explicit and natural in classes, and is just one of the things that classes do. Classes offer a more complete programming solution.

Customization via inheritance

Classes also support the OOP notion of inheritance: we can *extend* a class by redefining its attributes outside the class itself in new software components coded as subclasses. More generally, classes can build up namespace *hierarchies*, which define names to be used by objects created from classes in the hierarchy. This supports multiple customizable behaviors more directly than other tools.

Operator overloading

By providing special protocol methods, classes can define objects that respond to the sorts of *operations* we saw at work on built-in types. For instance, objects made with classes can be sliced, concatenated, indexed, and so on. Python provides hooks that classes can use to intercept and implement any built-in type operation.

At its base, the mechanism of OOP in Python largely boils down to just two bits of magic: a special first *argument* in functions (to receive the subject of a call) and inheritance attribute *search* (to support programming by customization). Other than this, the model is largely just functions that ultimately process built-in types. While not radically new, though, OOP adds an extra layer of structure that supports programming better than flat procedural models. Along with the functional tools we met earlier, it

represents a major abstraction step above computer hardware that helps us build more sophisticated programs.

OOP from 30,000 Feet

Before we dig into what this all means in terms of code, let's get a better handle on the general ideas behind OOP. If you've never done anything object-oriented in your life before now, some of the terminology in this chapter may seem a bit perplexing on the first pass. Moreover, the motivation for these terms may be elusive until you've had a chance to study the ways that programmers apply them in larger systems. OOP is as much an experience as a technology.

Attribute Inheritance Search

The good news is that OOP is much simpler to understand and use in Python than in some other languages like C++ or Java. As a dynamically typed scripting language, Python removes much of the syntactic clutter and complexity that clouds OOP in other tools. In fact, much of the OOP story in Python boils down to this expression:

`object.attribute`

We've been using this expression throughout the book to access module attributes, call methods of objects, and so on. When we say this to an object that is derived from a `class` statement, however, the expression kicks off a *search* in Python—it searches a tree of linked objects, looking for the first appearance of *attribute* that it can find. When classes are involved, the preceding Python expression effectively translates to the following in natural language:

Find the first occurrence of *attribute* by looking in *object*, then in all classes above it, from bottom to top and left to right.

In other words, attribute fetches are simply tree searches. The term *inheritance* is applied to it because objects lower in a tree inherit attributes attached to objects higher in that tree. As the search proceeds from the bottom up, in a sense, the objects linked into a tree are the union of all the attributes defined in all their tree parents, all the way up the tree.

In Python, this is all very literal: we really do build up trees of linked objects with code, and Python really does climb this tree at runtime searching for attributes every time we use the `object.attribute` expression. To make this more concrete, Figure 26-1 sketches an example of one of these trees.

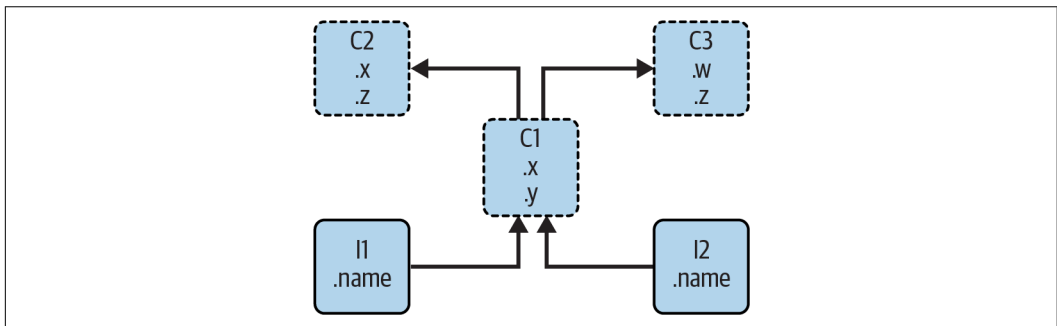


Figure 26-1. A class tree: instances (I1 and I2), a class (C1), and superclasses (C2 and C3)

In this figure, there is a tree of five objects labeled with variables, all of which have attached attributes, ready to be searched. More specifically, this tree links together three *class objects* (the ovals C1, C2, and C3) and two *instance objects* (the rectangles I1 and I2) into an inheritance-search tree. Notice that in the Python object model, classes and the instances you generate from them are two distinct object types:

Classes

Serve as instance factories. Their attributes provide behavior—data and functions—that is inherited by all the instances generated from them (e.g., a function to compute an employee’s salary from pay and hours).

Instances

Represent the concrete items in a program’s domain. Their attributes record data that varies per specific object (e.g., an employee’s pay rate and hours worked).

In terms of search trees, an instance inherits attributes from its class, and a class inherits attributes from all classes above it in the tree.

In [Figure 26-1](#), we can further categorize the ovals by their relative positions in the tree. We usually call classes higher in the tree (like C2 and C3) *superclasses*; classes lower in the tree (like C1) are known as *subclasses*. These terms refer to both relative tree positions and roles. Superclasses provide behavior shared by all their subclasses, but because the search proceeds from the bottom up, subclasses may override behavior defined in their superclasses by redefining superclass names lower in the tree.¹

As these last few words are really the crux of the matter of software *customization* in OOP, let’s expand on this concept. Suppose we build up the tree in [Figure 26-1](#), and then say this:

```
I2.w
```

Right away, this code invokes inheritance. Because this is an *object.attribute* expression, it triggers a search of the tree in [Figure 26-1](#)—Python will search for the attribute *w* by looking in I2 and above. Specifically, it will search the linked objects in this order:

```
I2, C1, C2, C3
```

and stop at the first attached *w* it finds (or raise an error if *w* isn’t found at all). In this case, *w* won’t be found until C3 is searched because it appears only in that object. In other words, *I2.w* resolves to *C3.w* by virtue of the automatic search. In OOP terminology, I2 “inherits” the attribute *w* from C3.

Ultimately, the two instances inherit four attributes from their classes: *w*, *x*, *y*, and *z*. Other attribute references will wind up following different paths in the tree. For example:

- *I1.x* and *I2.x* both find *x* in C1 and stop because C1 is lower than C2.
- *I1.y* and *I2.y* both find *y* in C1 because that’s the only place *y* appears.
- *I1.z* and *I2.z* both find *z* in C2 because C2 is further to the left than C3.
- *I2.name* finds *name* in I2 without climbing the tree at all.

Trace these searches through the tree in [Figure 26-1](#) to get a feel for how inheritance searches work in Python.

¹ In other literature and circles, you may also occasionally see the terms *base classes* and *derived classes* used to describe superclasses and subclasses, respectively. Most Python people and this book tend to use the latter terms.

The first item in the preceding list is perhaps the most important to notice—because C1 redefines the attribute `x` lower in the tree, it effectively *replaces* the version above it in C2. As you’ll see in a moment, such redefinitions are at the heart of software customization in OOP—by redefining and replacing the attribute, C1 effectively customizes what it inherits from its superclasses.

Classes and Instances

Although they are technically two separate object types in the Python model, the classes and instances we put in these trees are almost identical—each type’s main purpose is to serve as another kind of *namespace*—a package of variables, and a place where we can attach attributes. If classes and instances therefore sound like modules, they should; however, the objects in class trees also have automatically searched links to other namespace objects, and classes and instances correspond to statements and calls, respectively, not entire files.

The primary difference between classes and instances is that classes are a kind of *factory* for generating instances. For example, in a realistic application, we might have an `Employee` class that defines what it means to be an employee; from that class, we generate actual `Employee` instances. This is another difference between classes and modules—we only ever have one instance of a given module in memory (that’s why we have to reload a module to get its new code), but with classes, we can make as many unique instances as we need.

Operationally, classes will usually have functions attached to them (e.g., `computeSalary`), and the instances will have more basic data items used by the class’s functions (e.g., `hoursWorked`). In fact, the object-oriented model is not that different from the classic data-processing model of *programs* plus *records*—in OOP, instances are like records with “data,” and classes are the “programs” for processing those records. In OOP, though, we also have the notion of an inheritance hierarchy, which supports software customization better than earlier models.

Method Calls

In the prior section, we saw how the attribute reference `I2.w` in our example class tree was translated to `C3.w` by the inheritance search procedure in Python. Perhaps just as important to understand as the inheritance of attributes, though, is what happens when we try to call *methods*—functions attached to classes as attributes.

If this `I2.w` reference is a *function* call, what it really means is “call the `C3.w` function to process `I2`.” That is, Python will automatically map the call `I2.w()` into the call `C3.w(I2)`, passing in the instance as the first argument to the inherited function as the implied *subject* of the call.

In fact, whenever we call a function attached to a class in this fashion, an instance of the class is always implied. This implied subject is part of the reason we refer to this as an *object-oriented* model—there is always a subject object when an operation is run. In a more realistic example, we might invoke a method called `giveRaise` attached as an attribute to an `Employee` class; such a call has no meaning unless qualified with the employee to whom the raise should be given.

As you’ll see in more detail later, Python passes in the implied instance to a special first argument in the method, called `self` by strong convention. Methods go through this argument to process the subject of the call. As you’ll also learn later, methods can be called either through an instance—`pat.giveRaise()`—or through a class—`Employee.giveRaise(pat)`—and both forms serve purposes in our scripts. In fact, these calls illustrate both of the key ideas in OOP; to run a `pat.giveRaise()` method call, Python:

1. First looks up `giveRaise` from `pat`, by *inheritance*.
2. Then passes `pat` to the located `giveRaise` function, in the special `self` function *argument*.

When you run the call `Employee.giveRaise(pat)`, you're just performing both steps yourself.

This description is technically just the default case (Python has additional method types you'll meet later, called static and class methods), but it applies to the vast majority of the OOP code written in the language. To see how methods receive their subjects, though, we need to move on to some code.

Coding Class Trees

Although we are speaking in the abstract here, there is tangible code behind all these ideas, of course. We construct trees and their objects with `class` statements and class calls, which we'll explore in more detail later. In short, though:

- Each `class` statement generates a new class object.
- Each time a class is called, it generates a new instance object.
- Instances are automatically linked to the classes from which they are created.
- Classes are automatically linked to their superclasses according to the way we list them in parentheses in a `class` header line; the left-to-right order there gives the order in the tree.

To build the tree in [Figure 26-1](#), for example, we would run Python code of the sort in [Example 26-1](#). Like function definition, classes are normally coded in module files and are run during an import (the guts of the following `class` statements are omitted here for brevity, though `...` qualifies as a no-op statement per [Chapter 13](#) if run live).

Example 26-1. *classtree1.py*

```
class C2: ...           # Make class objects (ovals)
class C3: ...
class C1(C2, C3): ...   # Linked to superclasses - in this order

I1 = C1()               # Make instance objects (rectangles)
I2 = C1()               # Linked to their class
```

Here, we build the three class objects by running three `class` statements, and make the two instance objects by calling the class `C1` twice—as though it were a function (Python lumps classes and functions together as “callable” objects invoked with parentheses, though `def` requires header parentheses and `class` does not). The instances remember the class they were made from, and the class `C1` remembers its listed superclasses.

Technically, this example uses something called *multiple inheritance*, which simply means that a class has more than one superclass above it in the class tree—a useful technique when you wish to combine multiple tools. In Python, if there is more than one superclass listed in parentheses in a `class` statement (like `C1`'s here), their left-to-right order gives the order in which those superclasses will be searched for attributes by inheritance. The leftmost version of a name is used by default, though you can always choose a name by asking for it from the class it lives in (e.g., `C3.z`). The search also picks names to the right over higher duplicates, but we can safely ignore that for now.

Because of the way inheritance searches proceed, the object to which you attach an attribute turns out to be crucial—it determines the name’s *scope*. Attributes attached to instances pertain only to those single instances, but attributes attached to classes are shared by all their subclasses and instances. Later, we’ll study the code that hangs attributes on these objects in depth. As you’ll find, it’s all about where an *assignment* is run:

- Attributes are usually attached to classes by assignments made at the top level in `class` statement blocks, and not nested inside function `def` statements there.
- Attributes are usually attached to instances by assignments to the special argument passed to functions coded inside classes, called `self`.

For example, classes provide behavior for their instances with functions we create by coding `def` statements inside `class` statements. Because such nested `def` statements assign function names within the class, they wind up attaching attributes to the class object that will be inherited by all instances and subclasses—as is [Example 26-2](#), which lists changed parts in bold.

Example 26-2. *classtree2.py*

```
class C2: ...                # Make superclass objects
class C3: ...

class C1(C2, C3):            # Make and link class C1
    def setname(self, who):   # Assign name: C1.setname
        self.name = who      # Self is either I1 or I2

I1 = C1()                    # Make two instances
I2 = C1()

I1.setname('sue')            # Sets I1.name to 'sue'
I2.setname('bob')            # Sets I2.name to 'bob'
print(I1.name)               # Prints 'sue'
```

There’s nothing syntactically unique about `def` in this context. Operationally, though, when a `def` appears inside a class like this, it is usually known as a *method*, and it automatically receives a special first argument—called `self` by very strong convention—that provides a handle back to the instance to be processed. Any values you pass to the method yourself go to arguments after `self` (here, to `who`).²

Because classes are factories for multiple instances, their methods usually go through this automatically passed-in `self` argument whenever they need to fetch or set attributes of the particular instance being processed by a method call. In the preceding code, `self` is used to store a name in one of two instances.

Like simple variables, *attributes* of classes and instances are not declared ahead of time, but spring into existence the first time they are assigned values. When a method assigns to a `self` attribute, it creates or changes an attribute in an instance at the bottom of the class tree (i.e., one of the rectangles in [Figure 26-1](#)) because `self` automatically refers to the instance being processed—the subject of the call.

2 If you’ve ever used C++ or Java, you’ll recognize that Python’s `self` is much like these languages’ this pointer/reference, but `self` is always explicit in both headers and bodies of Python methods to make attribute accesses more obvious: a name has fewer possible meanings to consider, if it cannot be magically associated with a hidden object. Explicit is generally better.

In fact, because all the objects in class trees are just namespace objects, we can fetch or set any of their attributes by going through the appropriate names. Saying `C1.setname` is as valid as saying `I1.setname`, as long as the names `C1` and `I1` are in your code's scopes.

Operator Overloading

As currently coded, our `C1` class doesn't attach a `name` attribute to an instance until the `setname` method is called. Indeed, referencing `I1.name` before calling `I1.setname` would produce an undefined name error. If a class wants to guarantee that an attribute like `name` is always set in its instances, it more typically will fill out the attribute at construction time, as demoed by [Example 26-3](#).

Example 26-3. *classtree3.py*

```
class C2: ...                # Make superclass objects
class C3: ...

class C1(C2, C3):
    def __init__(self, who):   # Set name when constructed
        self.name = who      # Self is either I1 or I2

I1 = C1('sue')               # Sets I1.name to 'sue'
I2 = C1('bob')               # Sets I2.name to 'bob'
print(I1.name)               # Prints 'sue'
```

If it's either coded or inherited, Python automatically calls a method named `__init__` each time an instance is generated from a class. The new instance is passed in to the `self` argument of `__init__` as usual, and any values listed in parentheses in the class call go to arguments two and beyond. The effect here is to initialize instances when they are made, without requiring extra method calls.

The `__init__` method is known as the *constructor* because of when it is run. It's the most commonly used representative of a larger category called *operator-overloading* methods, which we'll explore in later chapters. Such methods are inherited in class trees as usual and have double underscores at the start and end of their names to make them distinct. Python runs them automatically when instances that support them appear in the corresponding operations, and they are mostly an alternative to using simple method calls. They're also optional: if omitted, the operations are not supported. If no `__init__` is present, class calls return an empty instance, without initializing it.

For example, a custom set intersection might be coded as a method named `intersect` called explicitly, or as a method named `__and__` that is called automatically by the `&` expression operator. Because the operator scheme makes instances look and feel more like built-in types, it allows some classes to provide a consistent and natural interface, and be compatible with code that expects a built-in type. Still, apart from the `__init__` constructor—which appears in most realistic classes—many programs may be better off with simpler named methods unless their objects are similar to built-ins. A `giveRaise` may make sense for an `Employee`, but an `&` might not.

OOP Is About Code Reuse

And that, along with a few syntax details, is most of the OOP story in Python. Of course, there's a bit more to it than just inheritance. For example, operator overloading is much more general than described so far—classes may also provide their own implementations of operations such as indexing, fetching attributes, printing, and more. By and large, though, OOP is about looking up attributes in trees with a special first argument in functions.

So why would we be interested in building and searching trees of objects? Although it takes some experience to see how, when used well, classes support code *reuse* in ways that other Python program components cannot. In fact, this is, by most accounts, their highest purpose. With classes, we code by customizing existing software, instead of either changing existing code in place or starting from scratch for each new project. This turns out to be a powerful paradigm in realistic programming.

At a fundamental level, classes are really just packages of functions and other names, much like modules. However, the automatic attribute inheritance search that we get with classes supports customization of software above and beyond what we can do with modules and functions. Moreover, classes provide a natural *structure* for code that packages and localizes both logic and names, and so aids in debugging.

To be fair, because methods are simply functions with a special first argument, we can mimic some of their behavior by manually passing subject objects to simple functions. The participation of methods in class inheritance, though, allows us to naturally extend and customize software by coding subclasses with new methods, rather than modifying code that already works. There is really no such concept with modules and functions.

Polymorphism and classes

As an abstract example, suppose you're assigned the task of implementing an employee database application. As a Python OOP programmer, you might begin by coding a general superclass that defines default behaviors common to all the kinds of employees in your organization (code in this section is hypothetical and partial):

```
class Employee:                                # General superclass
    def computeSalary(self): ...                # Common or default behaviors
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

Once you've coded this general behavior, you can specialize it for each specific kind of employee to reflect how the various types differ from the norm. That is, you can code subclasses that customize just the bits of behavior that vary per employee type; the rest of the employee types' behavior will be inherited from the more general class. For instance, if engineers have a unique salary computation rule (perhaps it's not hours times rate), you can replace just that one method in a subclass:

```
class Engineer(Employee):                      # Specialized subclass
    def computeSalary(self): ...                # Something custom here
```

Because the `computeSalary` version here appears lower in the class tree, it will replace (override) the general version in `Employee`. All other methods, though, are inherited from the superclass verbatim. You then create instances of the kinds of employee classes that the real employees belong to, to get the correct behavior:

```
sue = Employee()                               # Default behavior
bob = Employee()                               # Default behavior
pat = Engineer()                               # Custom salary calculator
```

Notice that you can make instances of any class in a tree, not just the ones at the bottom—the class you make an instance from determines the level at which the attribute search will begin, and thus which versions of the methods it will employ (pun accidental).

Ultimately, these three instance objects might wind up embedded in a larger container object—for instance, a list, dictionary, or an instance of another class—that represents a department or company using the composition idea mentioned at the start of this chapter. When you later ask for these

employees' salaries, they will be computed according to the classes from which the objects were made, due to the principles of the inheritance search:

```
company = [sue, bob, pat]           # A composite object
for emp in company:
    print(emp.computeSalary())       # Run this emp's version: default or custom
```

This is yet another instance of the idea of *polymorphism* introduced in [Chapter 4](#) and expanded in [Chapter 16](#). Recall that polymorphism means that the meaning of an operation depends on the object being operated on. That is, code shouldn't care about what an object *is*, only about what it *does*. Here, the method `computeSalary` is located by inheritance search in each object before it is called, per the object's class. The net effect is that we automatically run the correct version for the object being processed—default or custom. Trace the code to see why.

In other applications, polymorphism might also be used to *encapsulate* (i.e., abstract away) interface differences. For example, a program that processes data streams might be coded to expect objects with input and output methods, without caring what those methods actually do:

```
def processor(reader, converter, writer):
    while True:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

By passing in instances of subclasses that specialize the required `read` and `write` method interfaces for various data sources, we can reuse the `processor` function for any data source we need to use, both now and in the future:

```
class Reader:
    def other(self): ...           # Default behavior and tools

class FileReader(Reader):
    def read(self): ...           # Read from a local file

class SocketReader(Reader):
    def read(self): ...           # Read from a network socket

...and others...

processor(FileReader(...), converter, FileWriter(...))
processor(SocketReader(...), converter, FileWriter(...))
processor(FtpReader(...), converter, JsonWriter(...))
```

Moreover, because the internal implementations of those `read` and `write` methods have been factored into single locations, they can be changed without impacting code that uses them. The `processor` function might even be a class itself to allow the conversion logic of `converter` to be filled in by inheritance, and to allow readers and writers to be embedded by composition (you'll see how this works later in this part of the book).

Programming by customization

Once you get used to programming this way (by software customization), you'll find that when it's time to write a new program, much of your work may already be done—your task largely becomes one of mixing together existing superclasses that already implement the behavior required by your program. For example, someone else might have written the `Employee` and `Reader` classes in this section's examples for use in completely different programs. If so, you get all of that person's code “for free.”

In fact, in many application domains, you can fetch or purchase collections of superclasses, known as *frameworks*, that implement common programming tasks as classes, ready to be mixed into your applications. These frameworks might provide database interfaces, testing protocols, GUI toolkits, and so on. With frameworks, you often simply code a subclass that fills in a handful of expected methods; the framework classes higher in the tree do most of the work for you. Programming in such an OOP world is just a matter of combining and specializing already debugged code by writing subclasses of your own.

Of course, it takes a while to learn how to leverage classes to achieve such OOP utopia. In practice, object-oriented work also entails substantial design work to fully realize the code reuse benefits of classes. To this end, programmers catalog common OOP structures, known as *design patterns*, to help with design choices. The actual code you write to do OOP in Python, though, is so simple that it will not in *itself* pose an additional obstacle to your OOP quest. To see why, you'll have to move on to [Chapter 27](#).

Chapter Summary

We made an initial, abstract pass over classes and OOP in this chapter, taking in the big picture before we dive into syntax details. As we've seen, OOP is mostly about an argument named `self`, and a search for attributes in trees of linked objects called inheritance. Objects at the bottom of the tree inherit attributes from objects higher up in the tree—a feature that enables us to program by customizing code, rather than changing it or starting from scratch. When used well, this model of programming can cut development time radically.

The next chapter will begin to fill in the coding details behind the picture painted here. As we get deeper into Python classes, though, keep in mind that the OOP model in Python is very simple; as we've seen here, it's really just about looking up names in object trees and a special function argument. Before we move on, here's a quick quiz to review what we've covered here.

Test Your Knowledge: Quiz

1. What is the main point of OOP in Python?
2. Where does an inheritance search look for an attribute?
3. What is the difference between a class object and an instance object?
4. Why is the first argument in a class's method function special?
5. What is the `__init__` method used for?
6. How do you create a class instance?
7. How do you create a class?
8. How do you specify a class's superclasses?

Test Your Knowledge: Answers

1. OOP is about code *reuse*—you factor code to minimize redundancy, and program by customizing what already exists instead of changing code in place or starting from scratch.
2. An inheritance search looks for an attribute first in the instance object, then in the class the instance was created from, then in all higher superclasses, progressing from the bottom to the top

of the object tree, and from left to right (normally). The search stops at the first place the attribute is found. Because the lowest version of a name found along the way wins, class hierarchies naturally support customization by extension in new subclasses.

3. Both class and instance objects are namespaces—packages of variables that appear as attributes. The main difference between them is that classes are a kind of factory for creating multiple instances. Classes also support operator-overloading methods, which instances inherit, and treat any functions nested in the class as methods for processing instances.
4. The first argument in a class's method function is special because it always receives the instance object that is the implied subject of the method call. It's usually called `self` by convention. Because method functions always have this implied subject—and object context—by default, we say they are “object-oriented” (i.e., designed to process or change objects).
5. If the `__init__` method is coded or inherited in a class, Python calls it automatically each time an instance of that class is created. It's known as the *constructor* method; it is passed the new instance implicitly, as well as any arguments passed explicitly to the class name. It's also the most commonly used operator-overloading method. If no `__init__` method is present, instances simply begin life as empty namespaces.
6. You create a class instance by calling the class name as though it were a function; any arguments passed into the class name show up as arguments two and beyond in the `__init__` constructor method (if there is one). The new instance remembers the class it was created from for inheritance purposes.
7. You create a class by running a `class` statement; like function definitions, these statements normally run when the enclosing module file is imported (more on this in the next chapter).
8. You specify a class's superclasses by listing them in parentheses in the `class` statement, after the new class's name. The left-to-right order in which the classes are listed in the parentheses gives the left-to-right inheritance search order in the class tree.

Class Coding Basics

Now that we’ve talked about OOP in the abstract, it’s time to see how this translates to actual code. This chapter begins to fill in the syntax details behind the class model in Python.

If you’ve never been exposed to OOP in the past, classes can seem somewhat complicated if taken in a single dose. To make class coding easier to absorb, we’ll begin our detailed exploration of OOP by taking a first look at some basic classes in action in this chapter. We’ll expand on the details introduced here in later chapters of this part of the book, but in their basic form, Python classes are easy to understand.

In fact, classes have just three primary distinctions. At a base level, they are mostly just namespaces, much like the modules we studied in [Part V](#). Unlike modules, though, classes also have support for generating multiple objects, for namespace inheritance, and for operator overloading. Let’s begin our class statement tour by exploring each of these three distinctions in turn.

Classes Generate Multiple Instance Objects

To understand how the multiple objects idea works, you have to first understand that there are two kinds of objects in Python’s OOP model: *class* objects and *instance* objects. Class objects provide default behavior and are used to create instance objects. Instance objects are the tangible objects your programs process—each is a namespace in its own right but inherits (i.e., has automatic access to) names in the class from which it was created. Class objects come from statements, and instances come from calls; each time you call a class, you get a new instance of that class.

This object-generation concept is very different from most of the other program constructs we’ve seen so far in this book. In effect, classes are essentially *factories* for generating multiple instances. By contrast, only one copy of each module is ever imported into a single program. In fact, this is why `reload` works as it does, updating a single instance and shared object in place. With classes, each instance can have its own independent data, supporting multiple versions of the object that the class models.

In this role, class instances are similar to the per-call state of the *closure* (a.k.a. factory) functions of [Chapter 17](#), but this is a natural part of the class model, and state in classes is explicit attributes instead of implicit scope references. Moreover, this is just part of what classes do—they also support customization by inheritance, operator overloading, and multiple behaviors via methods. Hence, classes are a more complete programming tool, though OOP and *functional programming* are not mutually exclusive

paradigms. We may combine them by using functional tools in methods, by coding methods that are themselves generators, by writing user-defined iterators, and so on.

The following is a quick summary of the bare essentials of Python OOP in terms of its two object types. As you'll see, Python classes are in some ways similar to both `defs` and modules, but they may be quite different from what you're used to in other languages.

Class Objects Provide Default Behavior

When we run a `class` statement, we get a class object. Here's a rundown of the main properties of Python classes:

- **The `class` statement creates a class object and assigns it a name.** Just like the function `def` statement, the Python `class` is an executable statement. When reached and run, it generates a new class object and assigns it to the first name in the `class` header. Also, like `defs`, `class` statements typically run when the files they are coded in are first imported or run as a top-level script.
- **Assignments inside `class` statements make class attributes.** Just like in module files, top-level assignments within a `class` statement (not nested in a `def`) generate attributes in a class object. Technically, the `class` statement defines a local scope that *morphs* into the attribute namespace of the class object, just like a module's global scope. After running a `class` statement, class attributes may be accessed by name qualification: `class.name`.
- **Class attributes provide object state and behavior.** Attributes of a class object record state information and behavior to be shared by all instances created from the class. Most notably and commonly, function `def` statements nested inside a `class` generate *methods*, which process instances.

Instance Objects Are Concrete Items

When we call a class object, we get an instance object. Here's an overview of the key points behind class instances:

- **Calling a class object like a function makes a new instance object.** Each time a class is called, it creates and returns a new instance object. Instances represent material items in your program's domain.
- **Each instance object inherits class attributes and gets its own namespace.** Instance objects created from classes are new namespaces; they start out empty but inherit attributes that live in the class objects from which they were generated.
- **Assignments to attributes of `self` in methods make per-instance attributes.** Inside a class's method functions, the first argument (called `self` by convention) references the instance object being processed; assignments to attributes of `self` create or change data in the instance, not the class.

The end result is that classes define common, shared data and behavior, and generate instances. Instances reflect palpable application entities and record per-instance data that may vary per object.

A First Example

Let's turn to a real example to show how these ideas work in practice. To begin, let's define a class named `FirstClass` by running a Python `class` statement interactively in any REPL:


```
>>> class FirstClass:           # Define a class object
    def setdata(self, value):    # Define class's methods
        self.data = value       # self is the instance
    def display(self):
        print(self.data)        # self.data: per instance
```

We're working interactively here, but typically, such a statement would be run when the module file it is coded in is imported. Like functions created with `defs`, this class won't even exist until Python reaches and runs this statement.

Like all compound statements, the `class` starts with a header line that lists the class name, followed by a body of one or more nested and (usually) indented statements. Here, the nested statements are `defs`; they define functions that implement the behavior the class means to export.

As we learned in [Part IV](#), `def` is really an assignment. Here, it assigns function objects to the names `setdata` and `display` in the `class` statement's scope, and so generates attributes attached to the class—`FirstClass.setdata` and `FirstClass.display`. In fact, any name assigned at the top level of the class's nested block becomes an attribute of the class.

Functions inside a class are usually and traditionally called *methods*. They're coded with normal `defs`, and they support everything we've learned about functions already—they can have defaults, return values, yield items on request, and so on. But in a method function, the first argument automatically receives an implied instance object when called—the subject of the call. Let's create a couple of instances of our class to see how this works:

```
>>> x = FirstClass()           # Make two instances
>>> y = FirstClass()           # Each is a new namespace
```

By *calling* the class this way (notice the parentheses), we generate instance objects, which are just namespaces that have access to their classes' attributes. Properly speaking, at this point, we have three objects: two instances and a class. And really, we have three linked namespaces, as sketched in [Figure 27-1](#). In OOP terms, we say that `x` “is a” `FirstClass`, as is `y`—they both inherit names attached to the class.

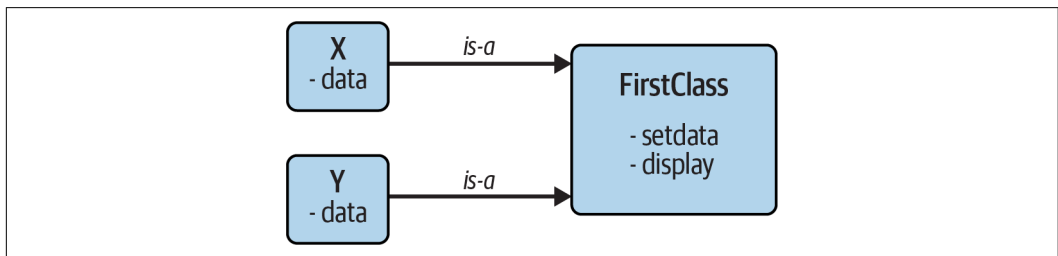


Figure 27-1. Classes and instances: namespaces in a class tree searched by inheritance

The two instances start out empty but have links back to the class from which they were generated. If we qualify an instance with the name of an attribute that lives in the class object, Python fetches the name from the class by inheritance search (unless it also lives in the instance):

```
>>> x.setdata('coding')        # Call methods: self is x
>>> y.setdata(3.14159)         # Runs: FirstClass.setdata(y, 3.14159)
```

Neither `x` nor `y` has a `setdata` attribute of its own, so to find it, Python follows the link from instance to class. And that's about all there is to inheritance in Python: it happens at attribute qualification time, and it just involves looking up names in linked objects—here, by following the *is-a* links in [Figure 27-1](#).

In the `setdata` function inside `FirstClass`, the value passed in is assigned to `self.data`. Within a method, `self`—the name given to the leftmost argument by convention—automatically refers to the instance being processed (`x` or `y` at this point), so the assignments store values in the instances’ namespaces, not the class’s. That’s how the data names in [Figure 27-1](#) are created.

Because classes can generate multiple instances, methods must go through the `self` argument to get to the instance to be processed. When we call the class’s `display` method to print `self.data`, we see that it’s different in each instance; on the other hand, the name `display` itself is the same in `x` and `y`, as it comes (is inherited) from the class:

```
>>> x.display()           # Runs: FirstClass.display(x)
coding
>>> y.display()           # self.data differs in each instance
3.14159
```

Notice that we stored different object types in the `data` member in each instance—a string and a floating-point number. As with everything else in Python, instance attributes (sometimes called *members*) are not predeclared and have no type constraints; they spring into existence the first time they are assigned values, just like simple variables. In fact, if we were to call `display` on one of our instances *before* calling `setdata`, we would trigger an undefined name error—the attribute named `data` doesn’t even exist in memory until it is assigned within the `setdata` method.

As another way to appreciate how dynamic this model is, consider that we can change instance attributes either inside the class itself, by assigning to `self` in methods, or *outside* the class, by assigning to an explicit instance object:

```
>>> x.data = 'hacking'     # Can get/set attributes
>>> x.display()           # Outside the class too
hacking
```

Although less common, we could even generate an entirely *new* attribute in the instance’s namespace by assigning to its name outside the class’s method functions:

```
>>> x.anothername = 'apps' # Can set new attributes here too!
```

This would attach a new attribute called `anothername`, which may or may not be used by any of the class’s methods, to the instance object `x`. Classes usually create all of the instance’s attributes by assignment to the `self` argument, but they don’t have to—programs can fetch, change, or create attributes on any objects to which they have references.

It usually doesn’t make sense to add data that the class cannot use, and it’s possible to prevent this with extra “privacy” code based on attribute-access operator overloading, as we’ll discuss elsewhere in this book (in [Chapters 30](#) and [39](#)). Still, free attribute access translates to less syntax, and there are cases where it’s even useful—for example, in coding data records of the sort we’ll build later in this chapter.

Classes Are Customized by Inheritance

Let’s move on to the second major distinction of classes. Besides serving as factories for generating multiple instance objects, classes also allow us to make changes by introducing new components (called *subclasses*), instead of changing existing components in place.

As we’ve seen, instance objects generated from a class inherit the class’s attributes. Python also allows classes to inherit from other classes, opening the door to coding *hierarchies* of classes that specialize behavior—by redefining attributes in subclasses that appear lower in the hierarchy, we override the more general definitions of those attributes higher in the tree. In effect, the further down the hierarchy

we go, the more specific the software becomes. Here, too, there is no parallel with modules, whose attributes live in a single, flat namespace that is not as amenable to customization.

In Python, instances inherit from classes, and classes inherit from superclasses. Here are the key ideas behind the machinery of attribute inheritance:

- **Superclasses are listed in parentheses in a class header.** To make a class inherit attributes from another class, just list the other class in parentheses in the new `class` statement's header line. The class that inherits is usually called a *subclass*, and the class that is inherited from is its *superclass*.
- **Classes inherit attributes from their superclasses.** Just as instances inherit the attribute names defined in their classes, classes inherit all of the attribute names defined in their superclasses. Python finds these names automatically when they're accessed if they don't exist in the subclasses.
- **Instances inherit attributes from all accessible classes.** Each instance gets names from the class it's generated from, as well as all of that class's superclasses. When looking for a name, Python checks the instance, then its class, then all superclasses above its class.
- **Each *object.attribute* reference invokes a new, independent search.** Python performs an independent search of the class tree for each attribute fetch expression. This includes references to instances and classes made outside `class` statements (e.g., `X.attr`), as well as references to attributes of the `self` instance argument in a class's method functions. That is, each `self.attr` expression in a method invokes a new search for `attr` in `self` and above.

The net effect—and the main purpose of all this searching—is that classes support factoring and customization of code better than any other language tool we've seen so far. On the one hand, they allow us to minimize code redundancy (and so reduce maintenance costs) by factoring operations into a single, shared implementation; on the other, they allow us to program by customizing what already exists, rather than changing it in place or starting from scratch.



Full inheritance disclosure: Strictly speaking, Python's *inheritance* is richer than described here, when we factor in “diamond” inheritance patterns and “meta-classes”—advanced topics we'll study later—but we can safely restrict our scope to instances and their classes, both at this point in the book and in most Python application code. We'll explore diamonds and the “MRO” inheritance search order that accommodates them in [Chapter 31](#), but our definition of inheritance won't be fully complete until [Chapter 40](#), because it regrettably requires meta-class info that's beyond almost all Python programmers' interest levels and pay grades (thankfully!).

A Second Example

To illustrate the role of inheritance, this next example builds on the previous one. First, we'll define a new class, `SecondClass`, that inherits all of `FirstClass`'s names and provides one of its own:

```
>>> class SecondClass(FirstClass):           # Inherits setdata
    def display(self):                         # Changes display
        print(f'Current value = {self.data}')
```

`SecondClass` defines the `display` method to print with a different format. By defining an attribute with the same name as an attribute in `FirstClass`, `SecondClass` effectively replaces the `display` attribute in its superclass.

Recall that inheritance searches proceed upward from instances to subclasses to superclasses, stopping at the first appearance of the attribute name that it finds. In this case, since the `display` name in `SecondClass` will be found before the one in `FirstClass`, we say that `SecondClass` *overrides* `FirstClass`'s `display`. Sometimes we call this act of replacing attributes by redefining them lower in the tree *overloading*.

The net effect here is that `SecondClass` specializes `FirstClass` by changing the behavior of the `display` method. On the other hand, `SecondClass` (and any instances created from it) still inherits the `setdata` method in `FirstClass` verbatim. Let's make a new instance to demonstrate:

```
>>> z = SecondClass()
>>> z.setdata('LP6e')      # Finds setdata in FirstClass
>>> z.display()            # Finds overridden method in SecondClass
Current value = "LP6e"
```

As before, we make a `SecondClass` instance object by calling it. The `setdata` call still runs the version in `FirstClass`, but this time the `display` attribute comes from `SecondClass` and prints a custom message. [Figure 27-2](#) sketches the namespaces involved.

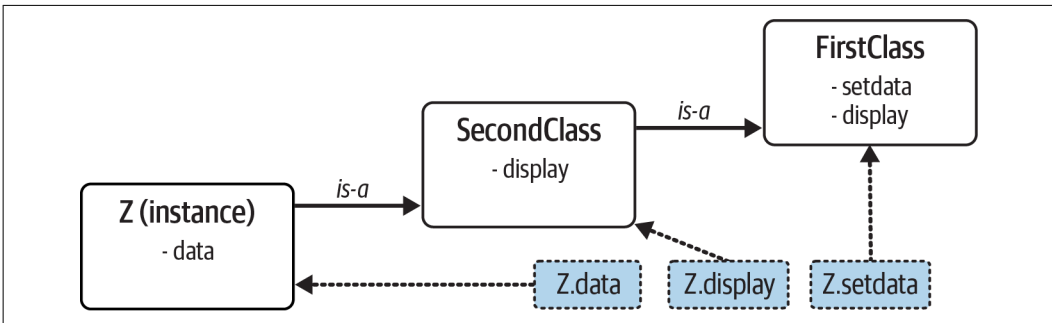


Figure 27-2. Specialization: overriding inherited names by redefining them in subclasses

Now, here's a crucial thing to notice about OOP: the specialization introduced in `SecondClass` is completely *external* to `FirstClass`. That is, it doesn't affect existing or future `FirstClass` objects, like the `x` from the prior example (assuming we're continuing the same REPL session):

```
>>> x.display()            # x is still a FirstClass instance (old message)
hacking
```

Rather than *changing* `FirstClass`, we *customized* it. Naturally, this is an artificial example, but as a rule, because inheritance allows us to make changes like this in external components (i.e., in subclasses), classes often support extension and reuse better than functions or modules can.

Classes Are Attributes in Modules

Before we move on, remember that there's nothing magic about a class name. It's just a variable assigned to an object when the `class` statement runs, and the object can be referenced with any normal expression. For instance, if our `FirstClass` were coded in a module file instead of being typed interactively, we could import it and use its name normally in a `class` header line:

```
from modulename import FirstClass      # Copy name into my scope
class SecondClass(FirstClass):         # Use class name directly
    def display(self): ...
```

Or equivalently:

```
import modulename                # Access the whole module
class SecondClass(modulename.FirstClass): # Qualify to reference
    def display(self): ...
```

Like everything else, class names always live within a module, so they must follow all the rules we studied in **Part V**. For example, more than one class can be coded in a single module file—like other statements in a module, class statements are run during imports to define names, and these names become distinct module attributes. More generally, each module may arbitrarily mix *any number* of variables, functions, and classes, and all names in a module behave the same way. The following hypothetical file demonstrates:

```
# names.py
var1 = 6                # names.var1
var2 = 3.12
def func1(): ...        # names.func1
def func2(): ...
class Cls1: ...          # names.Cls1
class Cls2: ...          # names.Cls2
```

This holds true even if the module and class happen to have the same name. For example, given the following imaginary file, *person.py*:

```
class person: ...
```

we need to go through the module to fetch the class as usual:

```
import person                # Import module
x = person.person()          # Class within module
```

Although this path may look redundant, it's required: `person.person` refers to the `person` class inside the `person` module. Saying just `person` gets the module, not the class, unless the `from` statement is used:

```
from person import person    # Get class from module
x = person()                  # Use class name
```

As with any other variable, we can never see a class in a file without first importing and somehow fetching it from its enclosing file. If this seems confusing, don't use the same name for a module and a class within it. In fact, common convention in Python recommends that class names should begin with an *uppercase* letter, and module names with a *lowercase* letter, to help make them more distinct (it's not required, but nearly common enough to be a rule):

```
import person                # Lowercase for modules
x = person.Person()          # Uppercase for classes
```

Also, keep in mind that although classes and modules are both namespaces for attaching attributes, they correspond to very different source code structures: a module reflects an entire *file*, but a class is a *statement* within a file. We'll say more about such distinctions later in this part of the book.

Classes Can Intercept Python Operators

Let's move on to the third and final major difference between classes and modules: operator overloading. In simple terms, *operator overloading* lets objects coded with classes intercept and respond to operations that work on built-in types: addition, slicing, printing, qualification, and so on. It's mostly just an automatic dispatch mechanism—expressions and other built-in operations route control to implementations in classes. Here, too, there is nothing similar in modules: modules can implement function calls, but not the behavior of expressions (*apart*, that is, from the odd special case added in Python 3.7 for

module `__getattr__` and `__dir__` functions covered in [Chapter 25](#), and bemoaned there as a confusing conflation with classes—for reasons you’re about to see for yourself).

Although we could implement all class behavior as normally named methods, operator overloading lets objects be more tightly integrated with Python’s object model. Moreover, because operator overloading makes our own objects act like built-ins, it tends to foster object interfaces that are more consistent and easier to learn, and it allows class-based objects to be processed by code written to expect a built-in object’s interface. Here is a quick rundown of the main ideas behind overloading operators:

- **Methods named with double underscores (`__X__`) are special hooks.** In Python classes, we implement operator overloading by providing specially named methods to intercept operations. The Python language defines a fixed and unchangeable mapping from each of these operations to a specially named method.
- **Such methods are called automatically when instances appear in built-in operations.** For instance, if an instance object inherits an `__add__` method, that method is called whenever the object appears in a `+` expression. The method’s return value becomes the result of the corresponding expression.
- **Classes may override most built-in type operations.** There are dozens of special operator-overloading method names for intercepting and implementing nearly every operation available for built-in object types. This includes expressions, but also basic operations like printing and object creation.
- **Most operator-overloading methods have no default, and none are required.** If a class does not define or inherit an operator-overloading method, it just means that the corresponding operation is not supported for the class’s instances. If there is no `__add__`, for example, `+` expressions raise exceptions. As you’ll learn later, a root class named `object` that’s an implicit superclass to every class does provide defaults for some `__X__` methods, but not for many (e.g., `object` has a default for print strings, but not `+`).

Importantly, operator overloading is an optional feature; it’s used primarily by people developing tools for other Python programmers, not by application developers. And, candidly, you probably *shouldn’t* use it just because it seems clever. Unless a class needs to mimic built-in object interfaces, it should usually stick to nonoperator method names whose calls are more explicit. Expressions like `*` and `+`, for example, may make sense for a numeric object like a matrix, but other code would generally serve its clients better with mnemonically named methods.

Because of this, we won’t go into details on every operator-overloading method available in Python in this learner’s book. Still, as previewed in the prior chapter, there is one operator-overloading method you are likely to see in almost every Python class: the `__init__` method, which is known as the *constructor* method and is used to initialize instance objects’ state. Pay special attention to this method, because `__init__`, along with the `self` argument and inheritance search, turns out to be a core requirement for reading and understanding most OOP code in Python.

A Third Example

On to another example. This time, we'll define a subclass of the prior section's `SecondClass` that implements three specially named attributes that Python will call automatically:

- `__init__` is run when a new instance object is created: `self` is the new `ThirdClass` object.¹
- `__add__` is run when a `ThirdClass` instance appears in a `+` expression.
- `__str__` is run when an object is printed (technically, when it's converted to its print string by the `str` built-in function or its Python-internals equivalent).

Our new subclass also defines a normally named method called `mul`, which changes the instance object in place. Here's the new subclass (copy-and-pasters: in REPLs, you may need to omit blank lines added here for clarity):

```
>>> class ThirdClass(SecondClass):           # Inherit from SecondClass
    def __init__(self, value):                 # On "ThirdClass(value)"
        self.data = value

    def __add__(self, other):                  # On "self + other"
        return ThirdClass(self.data + other)

    def __str__(self):                         # On "print(self)", "str()"
        return f'[ThirdClass: {self.data}]'

    def mul(self, other):                      # In-place change: named
        self.data *= other
```

`ThirdClass` “is a” `SecondClass`, so its instances inherit the customized display method from `SecondClass` of the preceding section. This time, though, `ThirdClass` creation calls pass an object to the `value` argument in the `__init__` constructor, where it is assigned to `self.data`. The net effect is that `ThirdClass` arranges to set the `data` attribute automatically at construction time, instead of requiring later `setdata` calls:

```
>>> a = ThirdClass(3)                       # __init__ called
>>> a.display()                             # Inherited method called
Current value = "3"
```

`ThirdClass` objects can also now show up in `+` expressions and `print` calls. For `+`, Python passes the instance object on the left to the `self` argument in `__add__` and the value on the right to `other`, as illustrated in [Figure 27-3](#); whatever `__add__` returns becomes the result of the `+` expression (more on its result in a moment):

```
>>> b = a + 3                               # __add__: makes a new instance
>>> b.display()                             # b has all ThirdClass methods
Current value = "6"
```

For `print`, Python passes the object being printed to `self` in `__str__`; whatever string this method returns is taken to be the print string for the object. With `__str__` (or its broader twin `__repr__`, which we'll leverage in the next chapter), we can use a normal `print` to display objects of this class, instead of

¹ Not to be confused with the `__init__.py` files in module packages! The method here is a class constructor function used to initialize the newly created instance, not a namespace for a module-package folder. See [Chapter 24](#) for more details.

calling the `display` method. As a contrast, the added `mul` method also changes the instance in place for a named call:

```
>>> print(b)           # __str__: returns display string
[ThirdClass: 6]
>>> a.mul(3)           # mul: changes instance in place
>>> print(a)           # Print this instance's data
[ThirdClass: 9]
```

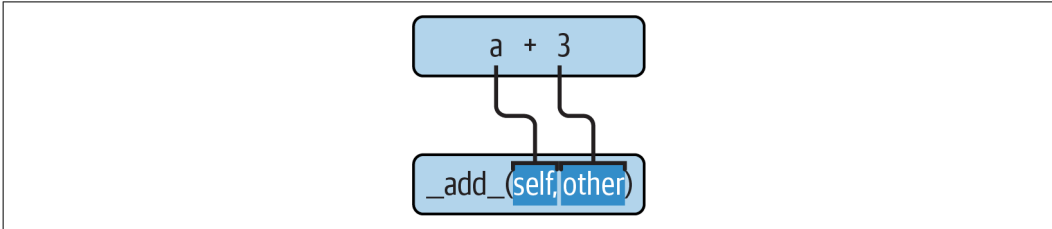


Figure 27-3. Operator overloading: class methods are run for operators and operations

Specially named methods such as `__init__`, `__add__`, and `__str__` are inherited by subclasses and instances, just like any other names assigned in a class. If they're not coded in a class, Python looks for such names in all its superclasses, as usual. As for all attributes, the lowest (most specific) version is used.

Operator-overloading method names are also not built-in or reserved words; they are just attributes that Python looks for when objects appear in various contexts. Python usually calls them automatically, but they may occasionally be called by your code as well. For example, the `__init__` method is often called manually to trigger required initialization steps in a superclass constructor, as you'll see in the next chapter.

Returning results—or not

Some operator-overloading methods like `__str__` require results, but others are more flexible. For example, notice how the `__add__` method makes and returns a *new* instance object of its class, by calling `ThirdClass` with the result value—which in turn triggers `__init__` to initialize the result. This is a common convention, and explains why `b` in the listing has a `display` method; it's a `ThirdClass` object too, because that's what `+` returns for this class's objects. This essentially propagates the object type.

By contrast, `mul` *changes* the current instance object in place, by reassigning the `self` attribute. We could overload the `*` expression to do the same with `__mul__`, but this would be too different from the behavior of `*` for built-in types such as numbers and strings, for which it always makes new objects. Per common practice, overloaded operators should work the same way that built-in operations do. Because operator overloading is really just an expression-to-method dispatch mechanism, though, you can interpret operators any way you like in your own classes. Also, stay tuned for [Chapter 30](#)'s related coverage of in-place operator methods (preview: `__imul__` handles `*=`).

Other operator-overloading methods

Although we won't cover every operator-overloading method in this book, we'll survey additional common operator-overloading techniques in [Chapter 30](#). Again, while this is an optional tool that doesn't apply to most application programs, `__str__` is not uncommon, and the `__init__` constructor method is a norm that will be present in most Python classes you'll come across. In fact, even though instance

attributes need not be predeclared in Python, you can usually find out which attributes an instance will have by inspecting its class's `__init__` method.

The World's Simplest Python Class

Despite the details of the `class` statement that we've begun to uncover in this chapter, you should keep in mind that the basic inheritance model behind classes is very simple—all it really involves is searching for attributes in trees of linked objects. In fact, we can create a class with nothing in it at all. The following statement makes a class with no attributes attached, an empty namespace object:

```
>>> class rec: pass                # Empty namespace object
```

We need the no-operation `pass` placeholder statement (discussed in [Chapter 13](#)) here because we don't have any methods to code. After we make the class by running this statement interactively, we can start attaching attributes to the class by assigning names to it completely outside of the original `class` statement:

```
>>> rec.name = 'Pat'               # Just objects with attributes
>>> rec.age = 40
```

And, after we've created these attributes by assignment, we can fetch them with the usual syntax. When used this way, a class is roughly similar to a “struct” in C, or a “record” in Pascal. It's basically an object with field names attached to it, and may be easier to code than alternatives like dictionaries:

```
>>> print(rec.name)               # Like a C struct or a record
Pat
```

Notice that this works even though there are *no instances* of the class yet; classes are objects in their own right, even without instances. In fact, they are just self-contained namespaces; as long as we have a reference to a class, we can set or change its attributes anytime we wish. Watch what happens when we do create two instances, though:

```
>>> x = rec()                     # Instances inherit class names
>>> y = rec()
```

These instances begin their lives as completely empty namespace objects. Because they remember the class from which they were made, though, they will obtain the attributes we attached to the class by inheritance:

```
>>> x.name, y.name                 # name is stored on the class only
('Pat', 'Pat')
```

Really, these instances have no attributes of their own; they simply fetch the `name` attribute from the class object where it is stored. If we do assign an attribute to an instance, though, it creates (or changes) the attribute in that object, and no other—crucially, attribute *references* kick off inheritance searches, but attribute *assignments* affect only the objects in which the assignments are made. Here, this means that `x` gets its own `name`, but `y` still inherits the `name` attached to the class above it:

```
>>> x.name = 'Sue'                # But assignment changes x only
>>> rec.name, x.name, y.name
('Pat', 'Sue', 'Pat')
```

Classes: Under the Hood

In fact, as we'll explore in more detail in [Chapter 29](#), the attributes of a namespace object are usually implemented as *dictionaries*, and class inheritance trees are, generally speaking, just dictionaries with links to other dictionaries. If you know where to look, you can see this explicitly.

For example, the `__dict__` attribute is the namespace dictionary for most class-based objects, much as in modules. Some classes may also—or instead—define attributes in `__slots__`, an advanced and seldom-used feature called *slots* with impacts that we’ll note along the way, but whose full coverage we’ll largely postpone until [Chapter 32](#). Normally, though, `__dict__` literally *is* the attribute namespace of an instance or class.

To illustrate, the following inspects the namespaces of objects in the prior section’s REPL session, filtering out built-ins in the class with a generator expression as we’ve done before leaving just the names we’ve assigned:

```
>>> list(key for key in rec.__dict__ if not key.startswith('__'))
['name', 'age']

>>> list(x.__dict__)
['name']
>>> list(y.__dict__)
[]
```

Here, the class’s namespace dictionary shows the `name` and `age` attributes we assigned to it, `x` has its own `name`, and `y` is still empty. Because of this model, an attribute can often be fetched by *either* dictionary indexing or attribute notation, but only if it’s present on the object in question—attribute notation kicks off inheritance *search*, but indexing looks in the single object *only*. As you’ll see later, both have valid roles, and the `dir` built-in collects inherited names:

```
>>> x.name, x.__dict__['name']          # Attributes present here are dict keys
('Sue', 'Sue')

>>> x.age                                # But attribute fetch checks classes too
40
>>> x.__dict__['age']                    # Indexing dict does not do inheritance
KeyError: 'age'

>>> x.__dict__                           # Object namespace versus inheritance
{'name': 'Sue'}
>>> [attr for attr in dir(x) if attr[:2] != '__']
['age', 'name']
```

In addition, to facilitate inheritance search on attribute fetches, each instance has a link to its class that Python creates for us—it’s called `__class__`, if you want to inspect it:

```
>>> x.__class__                          # Instance-to-class link
<class '__main__.rec'>
```

Classes also have a `__bases__` attribute, which is a tuple of references to a class’s superclass objects—in this example just the implicit and automatic object root class we’ll explore later:

```
>>> rec.__bases__                        # Class-to-superclasses link
(<class 'object'>,)
```

These two attributes are how class trees are literally represented in memory by Python. Internal details like these are not required knowledge—class trees are implied by the code you run, and their search is normally automatic—but they can often help demystify the model.

The main point here is that Python’s class model is extremely dynamic. Classes and instances are just linked namespace objects, with attributes created on the fly by assignment. Those assignments usually happen within the `class` statements you code, but they can occur anywhere you have a reference to one of the objects in the tree.

In fact, even *methods*, normally created by a `def` nested in a `class`, can be created completely independently of any class object. The following, for example, defines a simple function outside of any class that takes one argument:

```
>>> def upername(obj):
    return obj.name.upper()      # Still needs a self argument (obj)
```

There is nothing about a class here yet—it's a simple function, and it can be called as such at this point, provided we pass in an object `obj` with a `name` attribute, whose value in turn has an `upper` method—our class and instances happen to fit the expected interface and kick off string uppercase conversion:

```
>>> upername(rec), upername(x), upername(y)
('PAT', 'SUE', 'PAT')
```

If we assign this simple function to an attribute of our class, though, it becomes a *method*, callable through any instance, as well as through the class name itself as long as we pass in an instance manually—a technique we'll leverage further in the next chapter:²

```
>>> rec.method = upername      # Now it's a class's method!

>>> x.method()                 # Run method to process x
'SUE'

>>> y.method()                 # Same, but pass y to self
'PAT'

>>> rec.method(x)              # Can call through instance or class
'SUE'
```

Normally, classes are filled out by `class` statements, and instance attributes are created by assignments to `self` attributes in method functions. The point again, though, is that they don't have to be; OOP in Python really is mostly about looking up attributes in linked namespace objects.

Records Revisited: Classes Versus Dictionaries

Although the simple classes of the prior section are meant to illustrate class model basics, the techniques they employ can also be used for real work. For example, Chapters 8 and 9 showed how to use dictionaries, tuples, and lists to record properties of entities in our programs, generically called *records*. It turns out that classes can often serve better in this role—they package information like dictionaries, but can also bundle processing logic in the form of methods. For reference, here is an example for tuple- and dictionary-based records we used earlier in the book (using one of many dictionary coding techniques):

```
>>> rec = ('Pat', 40.5, ['dev', 'mgr'])      # Tuple-based record
>>> print(rec[0])
Pat

>>> rec = {}
>>> rec['name'] = 'Pat'                      # Dictionary-based record
>>> rec['age'] = 40.5                        # Or {...}, dict(n=v), etc.
>>> rec['jobs'] = ['dev', 'mgr']
```

2 In fact, this is one of the reasons the `self` argument *must* always be explicit in Python methods—because methods can be created as simple functions independent of a class, they need to include the implied instance argument explicitly. They can be called as either functions or methods, and Python can neither guess nor assume that a simple function might eventually become a class's method. The main reason for the explicit `self` argument, though, is to make the meanings of names more apparent: names not referenced through `self` are simple variables mapped to scopes, while names referenced through `self` with attribute notation are obviously instance attributes.

```
>>>
>>> print(rec['name'])
Pat
```

As we just saw, though, there are also multiple ways to do the same with classes. Perhaps the simplest is this—trading keys for attributes:

```
>>> class rec: pass

>>> rec.name = 'Pat'                                # Class-based record
>>> rec.age = 40.5
>>> rec.jobs = ['dev', 'mgr']
>>>
>>> print(rec.name)
Pat
```

This code has substantially less syntax than the dictionary equivalent. It uses an empty `class` statement to generate an empty namespace object, which we then fill out by assigning class attributes over time, as before. This works, but a new `class` statement will be required for each distinct record we will need. Perhaps more typically, we can instead generate *instances* of an empty class to represent each distinct entity:

```
>>> class rec: pass

>>> pers1 = rec()                                    # Instance-based records
>>> pers1.name = 'Bob'
>>> pers1.jobs = ['dev', 'mgr']
>>> pers1.age = 40.5
>>>
>>> pers2 = rec()
>>> pers2.name = 'Sue'
>>> pers2.jobs = ['dev', 'cto']
>>>
>>> pers1.name, pers2.name
('Bob', 'Sue')
```

Here, we make two records from the same class. Instances start out life empty, just like classes. We then fill in the records by assigning to attributes. This time, though, there are two separate objects, and hence two separate `name` attributes. In fact, instances of the same class don't even have to have the same set of attribute names; in this example, one has a unique `age` name. Instances really are distinct namespaces, so each has a distinct attribute dictionary. Although they are normally filled out consistently by a class's methods, they are more flexible than you might expect.

Finally, we might instead code a more full-blown class to implement the record *and* its processing—something that data-oriented dictionaries do not directly support:

```
>>> class Person:
    def __init__(self, name, jobs, age=None):        # class = data + logic
        self.name = name
        self.jobs = jobs
        self.age = age
    def info(self):
        return (self.name, self.jobs)

>>> rec1 = Person('Bob', ['dev', 'mgr'], 40.5)      # Construction calls
>>> rec2 = Person('Sue', ['dev', 'cto'])
>>>
>>> rec1.jobs, rec2.info()                          # Attributes + methods
(['dev', 'mgr'], ('Sue', ['dev', 'cto']))
```

This scheme also makes multiple instances, but the class is not empty this time: we've added *logic* (methods) to initialize instances at construction time and collect attributes into a tuple on request. The constructor imposes some consistency on instances here by always setting the `name`, `job`, and `age` attributes, even though the latter can be omitted when an object is made. Together, the class's methods and instance attributes create a package, which combines both *data* and *logic*.

We could further extend this code by adding logic to compute salaries, parse names, and so on. Ultimately, we might link the class into a larger hierarchy to inherit and customize an existing set of methods via the automatic attribute search of classes, or perhaps even store instances of the class in a file with Python object pickling to make them persistent. In fact, we *will*—in the next chapter, we'll expand on this analogy between classes and records with a more realistic running example that demonstrates class basics in action.

To be fair to other tools, in this form, the two preceding class construction calls more closely resemble *dictionaries* made all at once, but still seem less cluttered and provide extra processing methods. In fact, the class's construction calls more closely resemble [Chapter 9's named tuples](#)—which makes sense, given that named tuples really *are* classes with extra logic to map attributes to tuple offsets:

```
>>> rec = dict(name='Pat', jobs=['dev', 'mgr'], age=40.5)           # Dictionaries
>>> rec = {'name': 'Pat', 'jobs': ['dev', 'mgr'], 'age': 40.5}     # Ditto
>>> ...setup code...
>>> rec = Rec('Pat', ['dev', 'mgr'], 40.5)                         # Named tuples
```

In the end, although types like dictionaries and tuples are flexible, classes allow us to add *behavior* to objects in ways that built-in types and simple functions do not directly support. Although we can store functions in dictionaries too (e.g., under key `info` to mimic the class's method), using them to process implied instances is nowhere near as natural and structured as it is in classes, and key lookup has no notion of inheritance to enable customization.

But to vet the purported benefits of classes firsthand, you'll have to move ahead to the next chapter.

Chapter Summary

This chapter introduced the basics of coding classes in Python. We studied the syntax of the `class` statement and learned how to use it to build up a class inheritance tree. We also studied how Python automatically fills in the first argument in method functions, how attributes are attached to objects in a class tree by simple assignment, and how specially named operator-overloading methods intercept and implement built-in operations for our instances (e.g., expressions and printing).

Now that we've explored the mechanics of classes in Python, the next chapter turns to a larger and more realistic example that ties together much of what we've learned about OOP so far and introduces some new topics. After that, we'll continue our look at class coding, taking a second pass over the model to fill in some of the details that were omitted here to keep things simple. First, though, let's work through a quiz to review the basics we've covered so far.

Test Your Knowledge: Quiz

1. How are classes related to modules?
2. How are instances and classes created?
3. Where and how are class attributes created?

4. Where and how are instance attributes created?
5. What does `self` mean in a Python class?
6. How is operator overloading coded in a Python class?
7. When might you want to support operator overloading in your classes?
8. Which operator-overloading method is most commonly used?
9. What are the three key concepts required to understand Python OOP code?

Test Your Knowledge: Answers

1. Classes are always nested inside a module; they are attributes of a module object. Classes and modules are both namespaces, but classes correspond to statements (not entire files) and support the OOP notions of multiple instances, inheritance, and operator overloading (modules mostly do not). In a sense, a module is like a single-instance class, without inheritance, which corresponds to an entire file of code.
2. Classes are made by running class statements; instances are created by calling a class as though it were a function.
3. Class attributes are created by assigning attributes to a class object. They are normally generated by top-level assignments nested in a `class` statement—each name assigned in the `class` statement block becomes an attribute of the class object (technically, the `class` statement's local scope morphs into the class object's attribute namespace, much like a module). Class attributes can also be created, though, by assigning attributes to the class anywhere a reference to the class object exists—even outside the `class` statement.
4. Instance attributes are created by assigning attributes to an instance object. They are normally created within a class's method functions coded inside the `class` statement, by assigning attributes to the `self` argument (which is always the implied instance). Again, though, they may be created by assignment anywhere a reference to the instance appears, even outside the `class` statement. Usually, all instance attributes are initialized in the `__init__` constructor method; that way, later method calls can assume the attributes already exist.
5. `self` is the name commonly given to the first (leftmost) argument in a class's method function; Python automatically fills it in with the instance object that is the implied subject of the method call. This argument need not be called `self` (though this is a very strong convention); its position is what is significant. (Ex-C++ or Java programmers might prefer to call it `this` because in those languages that name reflects the same idea; in Python, though, this argument must always be explicit in method headers and code.)
6. Operator overloading is coded in a Python class with specially named methods; they all begin and end with double underscores to make them unique. These are not built-in or reserved names; Python just runs them automatically when an instance appears in the corresponding operation. Python itself defines the mappings from operations to special method names.
7. Operator overloading is useful to implement objects that resemble built-in types (e.g., sequences or numeric objects such as matrixes), and to mimic the built-in type interface expected by a piece of code. Mimicking built-in type interfaces enables you to pass in class instances that also have state information (i.e., attributes that remember data between operation calls). You generally shouldn't use operator overloading when a simple named method will suffice, though.

8. The `__init__` constructor method is the most commonly used; almost every class uses this method to set initial values for instance attributes and perform other startup tasks. The `__str__` method (and its `__repr__` sibling) is not uncommon, but not as much of a fixture as `__init__`.
9. The special `self` argument in method functions, the inheritance search for attributes, and the `__init__` constructor method are the cornerstones of OOP code in Python. If you get these, you should be able to read the text of most OOP Python code—apart from these, it's largely just packages of functions. In classes, `self` represents the automatic object argument, and `__init__` is commonly used to initialize instances.

A More Realistic Example

We'll dig into more class syntax details in the next chapter. Before we do, though, let's take a short detour to explore a realistic example of classes in action that's more practical than what we've seen so far. In this chapter, we're going to build a set of classes that do something concrete—recording and processing information about people. As you'll see, what we call *instances* and *classes* in Python programming can often serve the same roles as *records* and *programs* in more traditional terms. The main difference here is the customization that inheritance will enable.

Specifically, in this chapter we're going to code two classes:

- **Person**—a class that creates and processes information about people
- **Manager**—a customization of **Person** that modifies inherited behavior

Along the way, we'll make instances of both classes and test out their functionality. When we're done, this chapter will also show you a nice example use case for classes—we'll store our instances in a simple object-oriented database, to make them permanent. That way, you can use this code as a template for fleshing out a full-blown personal database of your own written entirely in Python.

Besides actual utility, though, our aim here is also *educational*: this chapter provides a tutorial on object-oriented programming in Python. Often, people grasp the last chapter's class syntax in the abstract but have trouble seeing how to get started when confronted with coding a new class from scratch. Toward this end, we'll take it one step at a time here, to help you learn the basics; we'll build up the classes gradually, so you can see how their features come together in complete programs.

In the end, our classes will still be relatively small in terms of code, but they will demonstrate *all* of the main ideas in Python's OOP model. Despite its syntax details, Python's class system really is largely just a matter of searching for an attribute in a tree of objects, along with a special first argument for functions.

Step 1: Making Instances

OK, so much for the design phase—let's move on to implementation. Our first task is to start coding the main class, **Person**. In your favorite text editor, open a new file for the code we'll be writing.

As noted in the prior chapter, it's a fairly strong convention in Python to begin module names with a lowercase letter and class names with an uppercase letter. Like the name of `self` arguments in methods, this is not required by the language, but it's so common that deviating might be confusing to people who later read your code. To conform, we'll call our new module file *person.py* and our class within it `Person`, as in [Example 28-1](#).

Example 28-1. person_1.py (start)

```
class Person:                                # Start a class
```

We're going to change this file as we go. To help you keep track of its variations, we'll append an *example number* to the filename in captions, and use that number both in the examples package and in launches and imports here. Changes will also be shown in bold font on each revision. The intent is to show mods to a single file, but books are linear.

As noted in the prior chapter, we can code any number of functions and classes in a single module file in Python, and this one's *person.py* name might not make much sense if we add unrelated components to it later. For now, we'll assume everything in it will be `Person`-related. It probably should be anyhow—as we've learned, modules tend to work best when they have a single, *cohesive* purpose.

Coding Constructors

Now, the first thing we want to do with our `Person` class is record basic information about people—to fill out record fields, if you will. Of course, these are known as instance object *attributes* in Python-speak, and they generally are created by assignment to `self` attributes in a class's method functions. The normal way to give instance attributes their first values is to assign them to `self` in the `__init__` *constructor method*, which contains code run automatically by Python each time an instance is created. Let's add one to our class, in [Example 28-2](#).

Example 28-2. person_2.py (add attribute initialization)

```
class Person:
    def __init__(self, name, job, pay):      # Constructor takes three arguments
        self.name = name                    # Fill out fields when created
        self.job = job                      # self is the new instance object
        self.pay = pay
```

This is a very common coding pattern: we pass in the data to be attached to an instance as arguments to the constructor method and assign them to `self` to retain them permanently. In OO terms, `self` is the newly created instance object, and `name`, `job`, and `pay` become *state information*—descriptive data saved on an object for later use. Although other techniques (such as enclosing scope reference closures) can save details, too, instance attributes make this very explicit and easy to understand.

Notice that the argument names appear *twice* here. This code might even seem a bit redundant at first, but it's not. The `job` argument, for example, is a local variable in the scope of the `__init__` function, but `self.job` is an attribute of the instance that's the implied subject of the method call. They are two different variables, which happen to have the same name. By assigning the `job` local to the `self.job` attribute with `self.job=job`, we save the passed-in `job` on the instance for later use. As usual in Python, where a name is assigned, or what object it is assigned to, determines what it means.

Speaking of arguments, there's really nothing magical about `__init__`, apart from the fact that it's called automatically when an instance is made, and has a special first argument. Despite its weird name, it's a

normal function and supports all the features of functions we’ve already covered. We can, for example, provide *defaults* for some of its arguments, so they need not be provided in cases where their values aren’t available or useful.

To demonstrate, let’s make the `job` argument optional—it will default to `None`, meaning the person being created is not (currently?) employed. If `job` defaults to `None`, we’ll probably want to default `pay` to `0`, too, for consistency (unless some of the people you know manage to get paid without having jobs). In fact, we have to specify a default for `pay` because according to Python’s syntax rules and [Chapter 18](#), any arguments in a function’s header after the first default must all have defaults, too. [Example 28-3](#) codes the mod.

Example 28-3. person_3.py (add constructor defaults)

```
class Person:
    def __init__(self, name, job=None, pay=0):          # Normal function args
        self.name = name
        self.job = job
        self.pay = pay
```

What this code means is that we’ll need to pass in a name when making `Persons`, but `job` and `pay` are now optional; they’ll default to `None` and `0` if omitted. The `self` argument, as usual, is filled in by Python automatically to refer to the instance object—assigning values to attributes of `self` attaches them to the new instance.

Testing as You Go

This class doesn’t do much yet—it essentially just fills out the fields of a new record—but it’s a real working class. At this point, we could add more code to it for more features, but we won’t do that yet. As you’ve probably begun to appreciate already, programming in Python is really a matter of *incremental prototyping*—you write some code, test it, write more code, test again, and so on. Because Python provides both an interactive session and nearly immediate turnaround after code changes, it’s more natural to test as you go than to write a huge amount of code to test all at once.

Before adding more features, then, let’s test what we’ve got so far by making a few instances of our class and displaying their attributes as created by the constructor. We could do this interactively, but as you’ve also probably surmised by now, interactive testing has its limits—it gets tedious to have to reimport modules and retype test cases each time you start a new testing session. More commonly, Python programmers use the interactive prompt for simple one-off tests but do more substantial testing by writing code at the bottom of the file that contains the objects to be tested, as in [Example 28-4](#).

Example 28-4. person_4.py (add incremental self-test code)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

bob = Person('Bob Smith')                # Test the class
sue = Person('Sue Jones', job='dev', pay=100000)  # Runs __init__ automatically
print(bob.name, bob.pay)                 # Fetch attached attributes
print(sue.name, sue.pay)                 # sue's and bob's attrs differ
```

Notice here that the `bob` object accepts the defaults for `job` and `pay`, but `sue` provides values explicitly. Also, note how we use *keyword arguments* when making `sue`; we could pass by position instead, but the keywords may help remind us later what the data is, and they allow us to pass the arguments in any left-to-right order we like. Again, despite its unusual name, `__init__` is a normal function, supporting everything you already know about functions—including both defaults and pass-by-name keyword arguments.

When this file runs as a script, the test code at the bottom makes two instances of our class and prints two attributes of each—`name` and `pay`:

```
$ python3 person_4.py
Bob Smith 0
Sue Jones 100000
```

You can also type this file's test code at Python's interactive prompt (assuming you import the `Person` class there first), but coding canned tests inside the module file like this makes it much easier to rerun them in the future.

Although this is fairly simple code, it's already demonstrating something important. Notice that `bob`'s `name` is not `sue`'s, and `sue`'s `pay` is not `bob`'s. Each is an independent record of information. Technically, `bob` and `sue` are both *namespace objects*—like all class instances, they each have their own independent copy of the state information created by the class. Because each instance of a class has its own set of `self` attributes, classes are a natural for recording information for multiple objects this way; just like built-in types such as lists and dictionaries, classes serve as a sort of *object factory*.

Other Python program structures, such as functions and modules, have no such concept. [Chapter 17](#)'s closure functions come close in terms of per-call state but don't have the multiple methods, inheritance, and larger structure we get from classes.

Using Code Two Ways

As is, the test code at the bottom of the file works, but there's a big catch—its top-level `print` statements run both when the file is run as a script and when it is imported as a module. This means if we ever decide to import the class in this file in order to use it somewhere else (and we will soon in this chapter), we'll see the output of its test code every time the file is imported. That's not very good software citizenship, though: client programs probably don't care about our internal tests and won't want to see our output mixed in with their own.

Although we could split the test code off into a separate file, it's often more convenient to code tests in the same file as the items to be tested. It would be better to arrange to run the test statements at the bottom *only* when the file is run for testing, not when the file is imported. As linear readers of this book have already learned, that's exactly what the module `__name__` check is designed for. [Example 28-5](#) shows what this addition looks like—simply add the required test and indent your self-test code.

Example 28-5. `person_5.py` (support both imports and run/tests)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

if __name__ == '__main__':
    bob = Person('Bob Smith')                                # When run for testing only
```

```
sue = Person('Sue Jones', job='dev', pay=100000)
print(bob.name, bob.pay)
print(sue.name, sue.pay)
```

Now, we get exactly the behavior we're after—running the file as a top-level script tests it because its `__name__` is `__main__`, but importing it as a library of classes later does not:

```
$ python3 person_5.py
Bob Smith 0
Sue Jones 100000

$ python3
>>> import person_5
>>>
```

When imported, the file now defines the class but does not use it. When run directly, this file creates two instances of our class as before, and prints two attributes of each; again, because each instance is an independent namespace object, the values of their attributes differ.

Step 2: Adding Behavior Methods

Everything looks good so far—at this point, our class is essentially a record *factory*; it creates and fills out fields of records (attributes of instances, in Pythonic terms). Even as limited as it is, though, we can still run some operations on its objects. Although classes add an extra layer of structure, they ultimately do most of their work by embedding and processing *core object types* like lists and strings. In other words, if you already know how to use Python's simple core objects, you already know much of the Python class story; classes are really just a minor structural extension.

For example, the `name` field of our objects is a simple string, so we can extract last names from our objects by splitting on spaces and indexing. These are all core-object operations, which work whether their subjects are embedded in class instances or not:

```
>>> name = 'Bob Smith'      # Simple string, outside class
>>> name.split()           # Extract last name
['Bob', 'Smith']
>>> name.split()[-1]       # Or [1], if always just two parts
'Smith'
```

Similarly, we can give an object a pay raise by updating its `pay` field—that is, by changing its state information in place with an assignment. This task also involves basic operations that work on Python's core objects, regardless of whether they are standalone or embedded in a class structure (formatting here masks any extraneous digits):

```
>>> pay = 100000           # Simple variable, outside class
>>> pay *= 1.10            # Give a 10% raise
>>> print(f'{pay:,.2f}')   # Or: pay = pay * 1.10, if you like to type
110,000.00                # Or: pay = pay + (pay * .10), if you really do
```

To apply these operations to the `Person` objects created by our script, simply do to `bob.name` and `sue.pay` what we just did to `name` and `pay`, as listed in [Example 28-6](#). The operations are the same, but the subjects are attached as attributes to objects created from our class.

Example 28-6. *person_6.py* (process embedded built-in objects)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.name.split()[-1])           # Extract object's last name
    sue.pay *= 1.10                        # Give this object a raise
    print(f'{sue.pay:,.2f}')
```

We’ve added the last three lines here; when they’re run, we extract bob’s last name by using basic string and list operations on his name field, and give sue a pay raise by modifying her pay attribute in place with basic number operations. In a sense, sue is also a *mutable* object—her state changes in place just like a list after an `append` call. Here’s the new version’s output when run as a top-level script:

```
$ python3 person_6.py
Bob Smith 0
Sue Jones 100000
Smith
110,000.00
```

The preceding code works as planned, but if you show it to a veteran software developer he or she will probably tell you that its general approach is not a great idea in practice. Hardcoding operations like these *outside* of the class can lead to maintenance problems in the future.

For example, what if you’ve hardcoded the last-name-extraction formula at many different places in your program? If you ever need to change the way it works (to support a new name structure, for instance), you’ll need to hunt down and update *every* occurrence. Similarly, if the pay-raise code ever changes (e.g., to require approval or database updates), you may have multiple copies to modify. Just finding all the appearances of such code may be problematic in larger programs—they may be scattered across many files and folders, split into individual steps, and so on. In a prototype like this, frequent change is almost guaranteed.

Coding Methods

What we really want to do here is employ a software design concept known as *encapsulation*—wrapping up operation logic behind interfaces, such that each operation is coded only once in our program. That way, if our needs change in the future, there is just one copy to update. Moreover, we’re free to change the single copy’s internals almost arbitrarily, without breaking the code that uses it.

In Python terms, we want to code operations on objects in a class’s *methods*, instead of littering them throughout our program. In fact, this is one of the things that classes are very good at—*factoring* code to remove *redundancy* and thus optimize maintainability. As an added bonus, turning operations into methods enables them to be applied to any instance of the class, not just those that they’ve been hardcoded to process.

This is all simpler in code than it may sound in theory. [Example 28-7](#) achieves encapsulation by moving the two operations from code outside the class to methods inside the class. While we’re at it, let’s change

our self-test code at the bottom to use the new methods we’re creating, instead of hardcoding operations.

Example 28-7. person_7.py (add methods to encapsulate operations)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
```

Behavior methods
self is implied subject
Must change here only
Use the new methods
instead of hardcoding

As we’ve learned, *methods* are simply normal functions that are attached to classes and designed to process instances of those classes. The instance is the subject of the method call and is passed to the method’s `self` argument automatically.

The transformation to the methods in this version is straightforward. The new `lastName` method, for example, simply does for `self` what the previous version hardcoded for `bob`, because `self` is the implied subject when the method is called. `lastName` also returns the result because this operation is a called function now; it computes a value for its caller to use arbitrarily, even if it is just to be printed. Similarly, the new `giveRaise` method just does for `self` what we did for `sue` before.

When run now, our file’s output is similar to before—we’ve mostly just *refactored* the code to allow for easier changes in the future (command lines that run the most recent mod are often omitted from here on for space):

```
Bob Smith 0
Sue Jones 100000
Smith Jones
110000
```

A few coding details are worth pointing out here. First, notice that `sue`’s pay is now still an *integer* after a pay raise—we convert the math result back to an integer by calling the `int` built-in within the method. Changing the value to either `int` or `float` is probably not a significant concern for this demo: integer and floating-point objects have the same interfaces and can be mixed within expressions. Still, we may need to address truncation and rounding issues in a real system—money probably is significant to Persons!

As covered in [Chapter 5](#), we might handle this by using the `round(N, 2)` built-in to round and retain cents, using the `decimal` type to fix precision, or storing monetary values as full floating-point numbers and displaying them with a formatting string to show cents as we did earlier. For now, we’ll simply truncate any cents with `int`.

Second, notice that we’re also printing sue’s last name this time—because the last-name logic has been encapsulated in a method, we get to use it on *any instance* of the class. As we’ve seen, Python tells a method which instance to process by automatically passing it in to the first argument, usually called `self`. Specifically:

- In the first call, `bob.lastName()`, `bob` is the implied subject passed to `self`.
- In the second call, `sue.lastName()`, `sue` goes to `self` instead.

Trace through these calls to see how the instance winds up in `self`—it’s a key concept. The net effect is that the method fetches the name of the implied subject each time. The same happens for `giveRaise`. We could, for example, give bob a raise by calling `giveRaise` for both instances this way, too. Unfortunately for bob, though, his zero starting pay will prevent him from getting a raise as the program is currently coded—nothing times anything is nothing, something we may want to address in a future 2.0 release of our software.

Finally, notice that the `giveRaise` method assumes that `percent` is passed in as a floating-point number between zero and one. That may be too radical an assumption in the real world (a 1,000% raise would probably be a bug for most of us!); we’ll let it pass for this prototype, but we might want to test or at least document this in a future iteration of this code. Stay tuned for a rehash of this idea in later chapters, where we’ll explore *function decorators* and Python’s `assert` statement as alternatives that can do the validity test for us automatically during development. In [Chapter 39](#), for example, we’ll write a tool that lets us validate with strange incantations like the following:

```
@rangetest(percent=(0.0, 1.0))           # Use decorator to validate
def giveRaise(self, percent):
    self.pay = int(self.pay * (1 + percent))
```

Step 3: Operator Overloading

At this point, we have a fairly full-featured class that generates and initializes instances, along with two new bits of behavior for processing instances in the form of methods. So far, so good.

As it stands, though, testing is still a bit less convenient than it needs to be—to trace our objects, we have to manually fetch and print *individual attributes* (e.g., `bob.name`, `sue.pay`). It would be nice if displaying an instance all at once actually gave us some useful information. Unfortunately, the default display format for an instance object isn’t very good—it displays the object’s class name and its address in memory (which is essentially useless in Python, except as a unique identifier).

To see this, change the last line in the latest version of our script to `print(sue)` so it displays the object as a whole. Here’s what you’ll get—as coded, the output says that sue is an object, but not much more:

```
Bob Smith 0
Sue Jones 100000
Smith Jones
<__main__.Person object at 0x104972630>
```

Providing Print Displays

Fortunately, it’s easy to do better by employing *operator overloading*—coding methods in a class that intercept and process built-in operations when run on the class’s instances. Specifically, we can make use of what are probably the second most commonly used operator-overloading methods in Python,

after `__init__`: the `__repr__` method we'll deploy here, and its `__str__` twin introduced in the preceding chapter.

These methods are run automatically every time an instance is converted to its print string. Because that's what `print` normally does, the net transitive effect is that printing an object displays whatever is returned by the object's `__str__` or `__repr__` method, if the object either defines one itself or inherits one from a superclass. Double-underscored names are inherited just like any other.

Technically, `__str__` is preferred by `print` and `str`, and `__repr__` is used both as a fallback for these, as well as by interactive echoes and nested appearances. Although the two can be used to implement different displays in different contexts (e.g., user- or developer-focused), coding `__repr__` alone suffices to give a single display in all cases. This allows clients to provide an alternative display with `__str__`, though this is a moot point for this demo.

The `__init__` constructor method we've already coded is, strictly speaking, operator overloading too—it is run automatically at construction time to initialize a newly created instance. Constructors are so common, though, that they almost seem like a special case. More focused methods like `__repr__` allow us to tap into specific operations and provide *specialized behavior* when our objects are used in those contexts.

Let's put this into code. **Example 28-8** extends our class to give a custom display that lists attributes when our class's instances are displayed as a whole, instead of relying on the less useful default display.

Example 28-8. person_8.py (add `__repr__` overload method for displays)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return f'Person: {self.name} ${self.pay:,}'

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
```

Notice that we're doing f-string formatting to build the display string in `__repr__` here, with comma separators for pay; at the bottom, classes use built-in operations like these to get their work done. Again, everything you've already learned about both built-in objects and user-defined functions applies to class-based code. Classes largely just add an additional layer of *structure* that packages functions and data together and supports extensions.

We've also changed our self-test code to print objects directly, instead of printing individual attributes. When run, the output is more uniform now; the "[...]" lines are returned by `__repr__`, run automatically by print operations:

```
[Person: Bob Smith $0]
[Person: Sue Jones $100,000]
Smith Jones
[Person: Sue Jones $110,000]
```

Step 4: Customizing Behavior by Subclassing

At this point, our class captures much of the OOP machinery in Python: it makes instances, provides behavior in methods, and even does a bit of operator overloading now to intercept print operations in `__repr__`. It effectively packages our data and logic together into a single, self-contained *software component*, making it easy to locate code and straightforward to change it in the future. By allowing us to encapsulate behavior, it also allows us to factor that code to avoid redundancy and its associated maintenance headaches.

The only major OOP concept it does not yet capture is *customization by inheritance*. In some sense, we're already doing inheritance, because instances inherit methods from their classes. To demonstrate the real power of OOP, though, we need to define a superclass/subclass relationship that allows us to extend our software and replace bits of inherited behavior. That's the main idea behind OOP, after all; by fostering a coding model based upon customization of work already done, it can dramatically cut development time when used well.

Coding Subclasses

As a next step, then, let's put OOP's methodology to use and customize our `Person` class by extending our software hierarchy. For the purpose of this tutorial, we'll define a subclass of `Person` called `Manager` that replaces the inherited `giveRaise` method with a more specialized version. Our new class begins as follows:

```
class Manager(Person):                                # Define a subclass of Person
```

This code means that we're defining a new class named `Manager`, which inherits from and may add customizations to the superclass `Person`. In plain terms, a `Manager` is almost like a `Person`... (admittedly, a very long journey for a very small joke), but `Manager` has a custom way to give raises.¹

For the sake of argument, let's assume that when a `Manager` gets a raise, it receives the passed-in percentage as usual, but also gets an extra bonus that defaults to 10%. For instance, if a `Manager`'s raise is specified as 10%, it will really get 20%. (Any relation to `Persons` living or dead is, of course, strictly coincidental.) Our new method begins as follows; because this redefinition of `giveRaise` will be closer in the class tree to `Manager` instances than the original version in `Person`, it effectively replaces, and thereby customizes, the operation. Recall that according to the inheritance search rules, the *lowest* version of the name wins (we'll add this code to the file in a moment):

```
class Manager(Person):                                # Inherit Person attrs
    def giveRaise(self, percent, bonus=.10):          # Redefine to customize
```

¹ And no offense to any managers in the audience, of course. This joke was once delivered at a Python class in New Jersey, and nobody laughed. The organizers later revealed that the attendees were all managers evaluating Python. Hence the silence.

Augmenting Methods: The Bad Way

Now, there are two ways we might code this `Manager` customization: a good way and a bad way. Let's start with the *bad way* since it might be a bit easier to understand. The bad way is to cut and paste the code of `giveRaise` in `Person` and modify it for `Manager`, like this:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus))    # Bad: cut and paste
```

This would work as advertised—when we later call the `giveRaise` method of a `Manager` instance, it would run this custom version, which tacks on the extra bonus. So what's wrong with something that runs correctly?

The problem here is a very general one: anytime you copy code with cut and paste, you essentially *double* your maintenance effort in the future. Think about it: because we copied the original version, if we ever have to change the way raises are given (and we probably will), we'll have to change the code in *two* places, not one. Although this is a small and artificial example, it's also representative of a universal issue—anytime you're tempted to program by copying code this way, you probably want to look for a better approach.

Augmenting Methods: The Good Way

What we really want to do here is somehow *augment* the original `giveRaise`, instead of replacing it altogether. The *good way* to do that in Python is by calling to the original version directly, with augmented arguments, like this:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)    # Good: augment original
```

This code leverages the fact that a class's method can always be called either through an *instance* (the usual way, where Python sends the instance to the `self` argument automatically) or through the *class* (the less common scheme, where you must pass the instance manually). In more symbolic terms, a normal method call of this form:

```
instance.method(args...)
```

is automatically translated by Python into this equivalent form:

```
class.method(instance, args...)
```

where the class containing the method to be run is determined by the inheritance search rule applied to the method's name. You can code *either* form in your script, but there is a slight asymmetry between the two—you must remember to pass along the instance manually if you call through the class directly.

The method always needs a subject instance one way or another, and Python provides it automatically only for calls made through an instance. For calls through the class name, you need to send an instance to `self` yourself; and for code inside a method like `giveRaise`, `self` already *is* the subject of the call, and hence the instance to pass along.

Calling through the class directly effectively subverts inheritance and kicks the call higher up the class tree to run a specific version. In our case, we can use this technique to invoke the default `giveRaise` in `Person`, even though it's been redefined at the `Manager` level. In fact, we *must* call through `Person` this way, because a `self.giveRaise()` inside `Manager`'s `giveRaise` code would loop—since `self` already is a

Manager, `self.giveRaise()` would resolve again to Manager’s `giveRaise`, and so on, *recursively*, until available memory is exhausted.

Call syntax aside, this “good” version may seem like a small difference in code, but it can make a huge difference for future *code maintenance*—because the `giveRaise` logic lives in just one place now (Person’s method), we have only one version to change in the future as needs evolve. And really, this form captures our intent more directly anyhow—we want to perform the standard `giveRaise` operation, but simply tack on an extra bonus. [Example 28-9](#) lists our entire module file with this step applied.

Example 28-9. person_9.py (add method customization in a subclass)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return f'[Person: {self.name} ${self.pay:,.}]'

class Manager(Person):
    def giveRaise(self, percent, bonus=.10):          # Redefine at this level
        Person.giveRaise(self, percent + bonus)      # Call Person's version

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    pat = Manager('Pat Jones', 'mgr', 50000)         # Make a Manager: __init__
    pat.giveRaise(.10)                                # Runs custom version
    print(pat.lastName())                             # Runs inherited method
    print(pat)                                         # Runs inherited __repr__
```

To test our Manager subclass customization, we’ve also added self-test code that makes a Manager, calls its methods, and prints it. When we make a Manager, we pass in a name, and an optional job and pay as before—because Manager had no `__init__` constructor, it inherits that in Person. Here’s the new version’s output:

```
[Person: Bob Smith $0]
[Person: Sue Jones $100,000]
Smith Jones
[Person: Sue Jones $110,000]
Jones
[Person: Pat Jones $60,000]
```

Everything looks good here: bob and sue are as before, and when pat the Manager is given a 10% raise, he or she really gets 20% (pay goes from \$50K to \$60K), because the customized `giveRaise` in Manager is run for this person only. Also notice how printing pat as a whole at the end of the test code displays the nice format defined in Person’s `__repr__`: Manager objects get this, `lastName`, and the `__init__` constructor method’s code “for free” from Person, by inheritance.

The super Alternative

To extend inherited methods, the examples in this chapter simply call the original through the superclass name: `Person.giveRaise(...)`. This is the explicit, traditional, and simplest scheme in Python, and the one used in most of this book. It also follows the same pattern we'll code to access *class attributes*: names attached to a class and shared by all instances, introduced ahead. Methods are just callable class attributes.

Java programmers may especially be interested to know that Python also has a `super` built-in function that allows calling back to a superclass's methods more generically. In the custom `giveRaise` of [Example 28-9](#)'s `Manager` class, both of the following would work the same way:

```
Person.giveRaise(self, percent + bonus)    # Explicit, general use
super().giveRaise(percent + bonus)         # Implicit, special case
```

The `super` built-in, however, relies on unusual semantics; works unevenly with Python's operator overloading; and does not always mesh well with multiple inheritance, where a single method call may not suffice. It's a special-case tool with a single role, which is redundant with general class-name references.

In its defense, the `super` call has a valid role too—cooperative same-named method dispatch in multiple inheritance trees—but this relies on the “MRO” class ordering of [Chapter 31](#), which many find artificial; requires universal deployment to be used reliably; does not fully support method replacement and varying argument lists; and to many observers seems an esoteric solution to a role that is rare in real Python code.

Because of these downsides, this book prefers to call superclasses by explicit name instead of `super`, recommends the same policy for newcomers, and defers presenting `super` until [Chapter 32](#). You're of course free to draw your own conclusions there, but `super` is usually best had after you learn the simpler and more explicit ways of achieving the same goals in Python, especially if you're new to OOP. Topics like cooperative multiple-inheritance dispatch are a lot to digest—for beginners and experts alike.

And to any Java programmers in the audience: try resisting the temptation to use Python's `super` until you've had a chance to study its subtle implications. This call is benign in single-inheritance of the sort you're used to, but once you step up to Python's multiple inheritance, `super` is not what you think it is, and more than you probably expect. The class it picks can vary per context, and may not even be a superclass at all!

Polymorphism in Action

To make this acquisition of inherited behavior even more striking, add the following code at the end of our file temporarily (this is file `person_9_plus.py` in the examples package, but doesn't warrant a full listing or caption here):

```
if __name__ == '__main__':
    ...
    print('--All three--')
    for obj in (bob, sue, pat):
        obj.giveRaise(.10)
        print(obj)
# Process objects generically
# Run this object's giveRaise
# Run the common __repr__
```

Here's the resulting output, showing just its new parts (and the accumulated effects of two raises per person):

```
...
--All three--
[Person: Bob Smith $0]
[Person: Sue Jones $121,000]
[Person: Pat Jones $72,000]
```

In the added code, object is *either* a Person or a Manager, and Python runs the appropriate `giveRaise` automatically—our original version in `Person` for bob and sue, and our customized version in `Manager` for pat. Trace the method calls yourself to see how Python selects the right `giveRaise` method for each object.

This is just Python's notion of *polymorphism*, which we met earlier in the book, at work again—what `giveRaise` does depends on what you do it to. Here, it's made all the more obvious when it selects from code we've written ourselves in classes. The practical effect in this code is that sue gets another 10% but pat gets another 20% because `giveRaise` is dispatched based on the object's type. As we've seen, polymorphism is at the heart of Python's flexibility. Passing any of our three objects to a function that calls a `giveRaise` method, for example, would have the same effect: the appropriate version would be run automatically, depending on which type of object was passed.

On the other hand, printing runs the *same* `__repr__` for all three objects, because it's coded just once in `Person`. `Manager` both specializes and applies the code we originally wrote in `Person`. Although this example is small, it's already leveraging OOP's talent for code customization and reuse; with classes, this almost seems automatic at times.

Inherit, Customize, and Extend

In fact, classes can be even more flexible than our example implies. In general, classes can *inherit*, *customize*, or *extend* existing code in superclasses. For example, although we're focused on customization here, we can also add unique methods to `Manager` that are not present in `Person`, if `Managers` require something different. The following abstract snippet illustrates. Here, `giveRaise` redefines a superclass's method to customize it, but `somethingElse` defines something new to extend:

```
class Person:
    def lastName(self): ...
    def giveRaise(self): ...
    def __repr__(self): ...

class Manager(Person):
    def giveRaise(self, ...): ...      # Inherit
    def somethingElse(self, ...): ...  # Customize
                                     # Extend

pat = Manager()
pat.lastName()      # Inherited verbatim
pat.giveRaise()     # Customized version
pat.somethingElse() # Extension here
print(pat)          # Inherited overload method
```

Extra methods like this code's `somethingElse` augment the existing software and are available on `Manager` objects only, not on `Persons`. For the purposes of this tutorial, however, we'll limit our scope to customizing some of `Person`'s behavior by redefining it, not adding to it.

OOP: The Big Idea

As is, our code may be small, but it's fairly functional. And really, it already illustrates the main point behind OOP in general: in OOP, we program by *customizing* what has already been done, rather than copying or changing existing code. This isn't always an obvious win to newcomers at first glance, especially given the extra coding requirements of classes. But overall, the programming style implied by classes can cut development time radically compared to other approaches.

For instance, in our example we could theoretically have implemented a custom `giveRaise` operation without subclassing, but none of the other options yield code as optimal as ours:

- Although we could have simply coded `Manager` *from scratch* as new, independent code, we would have had to reimplement all the behaviors in `Person` that are the same for `Managers`.
- Although we could have simply *changed* the existing `Person` class in place for the requirements of `Manager`'s `giveRaise`, doing so would break code that still needs the original `Person` behavior.
- Although we could have simply *copied* the `Person` class in its entirety, renamed the copy to `Manager`, and changed its `giveRaise`, doing so would introduce code redundancy that would double our work in the future—changes made to `Person` in the future would not be picked up automatically, but would have to be manually propagated to `Manager`'s code. As usual, the cut-and-paste approach may seem quick now, but it doubles your work in the future.

The *customizable hierarchies* we can build with classes provide a much better solution for software that will evolve over time. No other tools in Python support this development mode. Because we can tailor and extend our prior work by coding new subclasses, we can leverage what we've already done, rather than starting anew each time, breaking what already works, or introducing multiple redundant copies. When done right, OOP is a powerful ally.

Step 5: Customizing Constructors, Too

Our code works as it is, but if you study the current version closely, you may be struck by something a bit odd—it seems pointless to have to provide a `mgr` job name for `Manager` objects when we create them: this is already implied by the class itself. It would be better if we could somehow fill in this value automatically when a `Manager` is made.

The trick we need to improve on this turns out to be the *same* as the one we employed in the prior section: we want to customize the constructor logic for `Managers` in such a way as to provide a job name automatically. In terms of code, we want to redefine an `__init__` method in `Manager` that provides the `mgr` string for us. And as in `giveRaise` customization, we also want to run the original `__init__` in `Person` by calling through the class name, so it still initializes our objects' state information attributes.

The mods in [Example 28-10](#) will do the job—we've coded the new `Manager` constructor and changed the call that creates `pat` to not pass in the `mgr` job name.

Example 28-10. `person_10.py` (add constructor customization in a subclass)

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
```

```

        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return f'[Person: {self.name} ${self.pay:,}]'

class Manager(Person):
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    pat = Manager('Pat Jones', 50000)
    pat.giveRaise(.10)
    print(pat.lastName())
    print(pat)

```

Again, we're using the same technique to augment the `__init__` constructor here that we used for `giveRaise` earlier—running the superclass version by calling through the class name directly and passing the `self` instance along explicitly. Although the constructor has a strange name, the effect is identical. Because we need `Person`'s construction logic to run too (to initialize instance attributes), we really have to call it this way; otherwise, instances would not have any attributes attached.

Calling superclass constructors from redefinitions this way turns out to be a very common coding pattern in Python. By itself, Python uses inheritance to look for and call only *one* `__init__` method at construction time—the *lowest* one in the class tree. If you need higher `__init__` methods to be run at construction time (and you usually do), you must call them manually, and usually through the superclass's name. The upside to this is that you can be explicit about which argument to pass up to the superclass's constructor and can choose to *not* call it at all: not calling the superclass constructor allows you to replace its logic altogether, rather than augmenting it.

The output of this file's self-test code is the same as before—we haven't changed what this example does, we've simply restructured it to get rid of some logical redundancy:

```

$ python3 person_10.py
[Person: Bob Smith $0]
[Person: Sue Jones $100,000]
Smith Jones
[Person: Sue Jones $110,000]
Jones
[Person: Pat Jones $60,000]

```

Per the sidebar “[The super Alternative](#)” on page 691, an implicit `super().__init__(...)` sans `self` would run the superclass constructor too, and this is a common role for this built-in in single-inheritance class trees. For all the reasons noted earlier, though, let's stick to the explicit and general for now.

OOP Is Simpler Than You May Think

In this complete form, and despite their relatively small sizes, our classes capture nearly all the important concepts in Python's OOP machinery:

- Instance creation—filling out instance attributes
- Behavior methods—encapsulating logic in a class's methods
- Operator overloading—providing behavior for built-in operations like printing
- Customizing behavior—redefining methods in subclasses to specialize them
- Customizing constructors—adding initialization logic to superclass steps

Most of these concepts are based upon just three simple ideas: the inheritance search for attributes in object trees, the special `self` argument in methods, and operator overloading's automatic dispatch to methods.

Along the way, we've also made our code easy to change in the future, by harnessing the class's propensity for factoring code to reduce *redundancy*. For example, we wrapped up logic in methods and called back to superclass methods from extensions to avoid having multiple copies of the same code. Most of these steps were a natural outgrowth of the structuring power of classes.

By and large, that's all there is to OOP in Python. Classes certainly can become larger than this, and there are some more advanced class concepts, such as decorators and metaclasses, which we will explore in later chapters. In terms of the basics, though, our classes already do it all. In fact, if you've grasped the workings of the classes we've written, most OOP Python code should now be within your reach.

Other Ways to Combine Classes: Composites

Having said that, it should be noted that although the basic mechanics of OOP are simple in Python, some of the art in larger programs lies in the way that classes are put together. We're focusing on *inheritance* in this tutorial because that's the mechanism the Python language provides, but programmers sometimes combine classes in other ways, too.

For example, a common coding pattern involves nesting objects inside each other to build up *composites*. We'll explore this pattern in more detail in [Chapter 31](#), which is really more about design than about Python. As a quick example, though, we could use this composition idea to code our `Manager` extension by *embedding* a `Person`, instead of inheriting from it.

The alternative `Manager` in [Example 28-11](#) does so in part by using the `__getattr__` operator-overloading method to intercept undefined attribute fetches, and the `getattr` built-in to *delegate* them to the embedded object:

- The `__getattr__` method isn't covered in full until [Chapter 30](#), but its usage is simple enough to employ here—it's automatically run for all fetches of undefined attributes (those not found in the instance by normal inheritance search). And yes, this is the same name used for the nonclass module-attributes hook of [Chapter 25](#).
- The `getattr` call was introduced in [Chapter 25](#)—it's the same as `X.Y` attribute fetch notation and thus performs inheritance, but the attribute name `Y` is evaluated to yield a string, not a name interpreted literally.

More specifically here, the constructor makes the embedded object with job name; `giveRaise` customizes by changing the argument passed along to the embedded object; and `lastName` triggers `__getattr__` to reroute to the embedded object. In effect, `Manager` becomes a controller layer that passes calls *down* to the embedded object, rather than *up* to superclass methods.

Example 28-11. person-composite.py (embedding-based Manager alternative)

```
from person_10 import Person                                # Example 28-10's Person

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)             # Embed a Person object

    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)             # Intercept and delegate

    def __getattr__(self, attr):
        return getattr(self.person, attr)                  # Delegate all other attrs

    def __repr__(self):
        return str(self.person)                             # Must overload again per ahead

if __name__ == '__main__':
    pat = Manager('Pat Jones', 50000)                       # Embed a Person
    pat.giveRaise(.10)                                       # Run Manager.giveRaise
    print(pat.lastName())                                    # Delegate to embedded
    print(pat)                                                # Run Manager.__repr__
```

The output of this version tests just its `Manager` class (the imported [Example 28-10](#) tests its own code). As before, a 10% raise is munged to 20% by the custom `giveRaise` here, but other calls are handled by the embedded `Person`, by either direct calls or generic attribute rerouting:

```
$ python3 person-composite.py
Jones
[Person: Pat Jones $60,000]
```

The more important point here is that this `Manager` alternative is representative of a general coding pattern usually known as *delegation*—a composite-based structure that manages a wrapped object and propagates method calls to it.

This pattern works in our example, but it requires about twice as much code and is less well suited than inheritance to the kinds of direct customizations we meant to express. In fact, reasonable Python programmers would almost certainly not code this example this way in practice. `Manager` isn't really a `Person` here, so we need extra code to manually dispatch method calls to the embedded object; operator-overloading methods like `__repr__` must be redefined for reasons explained in the sidebar “[Delegating Built-ins—or Not](#)” on page 698; and adding new `Manager` behavior is less straightforward since state information is one level removed.

Still, *object embedding*, and design patterns based upon it, can be a good fit when embedded objects require more limited interaction with the container than direct customization implies. A controller layer, or *proxy*, like this alternative `Manager`, for example, might come in handy if we want to adapt a class to an expected interface it does not support, or trace or validate calls to another object's methods (indeed, we will use a nearly identical coding pattern when we study *class decorators* later in the book).

Moreover, a `Department` class prototype like that in [Example 28-12](#) could *aggregate* other objects in order to treat them as a set.

Example 28-12. person-department.py (aggregate embedded objects into a composite)

```
from person_10 import Person, Manager          # Example 28-10's classes

class Department:
    def __init__(self, *args):
        self.members = list(args)              # Manage an objects list

    def addMember(self, person):
        self.members.append(person)

    def giveRaises(self, percent):               # Apply methods to all objects
        for person in self.members:
            person.giveRaise(percent)

    def showAll(self):                           # Display all nested objects
        for person in self.members:
            print(person)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    pat = Manager('Pat Jones', 50000)

    development = Department(bob, sue)          # Embed objects in a composite
    development.addMember(pat)
    development.giveRaises(.10)                 # Runs embedded objects' giveRaise
    development.showAll()                       # Runs embedded objects' __repr__
```

When run, the department's `showAll` method lists all of its contained objects after updating their state in true polymorphic fashion with `giveRaises`:

```
$ python3 person-department.py
[Person: Bob Smith $0]
[Person: Sue Jones $110,000]
[Person: Pat Jones $60,000]
```

Interestingly, this code uses both inheritance *and* composition—`Department` is a composite that embeds and controls other objects to aggregate, but the embedded `Person` and `Manager` objects themselves use inheritance to customize. As another example, a GUI might similarly use *inheritance* to customize the behavior or appearance of labels and buttons, but also *composition* to build up larger packages of embedded widgets, such as input forms, calculators, and text editors. The class structure to use depends on the objects you are trying to model—in fact, the ability to model real-world entities this way is one of OOP's strengths.

Design issues like composition are explored in [Chapter 31](#), so we'll postpone further investigations for now. But again, in terms of the basic mechanics of OOP in Python, our `Person` and `Manager` classes already tell the entire story. Now that you've mastered the basics of OOP, though, developing general tools for applying it more easily in your scripts is often a natural next step—and the topic of the next section.

Delegating Built-ins—or Not

Notice how the delegation-based `Manager` class of [Example 28-11](#) redefines the `__repr__` run by `print`, instead of allowing it to be caught by `__getattr__` like other attributes. In general, classes cannot intercept and delegate operator-overloading method attributes used by *built-in operations* without redefining them this way. Although we know that `__repr__` is the only such name used in our specific example, this is a broader issue for delegation-based classes.

Recall that built-in operations like printing and addition implicitly invoke operator-overloading methods such as `__repr__` and `__add__`. Built-in operations like these, however, do *not* route their implicit attribute fetches through generic attribute managers: neither `__getattr__` (run for undefined attributes, and used here) nor `__getattribute__` (run for all attributes, and covered later) is invoked. Moreover, the implicit object superclass at the top of all class trees defines defaults that can preclude `__getattr__` too.

Hence, `Manager` must redefine `__repr__` redundantly, in order to route printing to the embedded `Person` object. Comment out this class's `__repr__` method to see this live—the `Manager` instance then prints with a default instead of our custom display, because `__getattr__` is never run for printing:

```
$ python3 person-composite.py
Jones
<__main__.Manager object at 0x10a292c30>
```

Technically, built-in operations begin their implicit search for method names at the *class*, skipping the instance entirely. By contrast, explicit by-name attribute fetches are always routed to the *instance* first. Classes also inherit a default `__repr__` from their automatic object superclass that would prevent `__getattr__` from running too, but the more absolute `__getattribute__` doesn't intercept the name either, and other methods *not* in object also won't be caught by `__getattr__`, like `__add__` for +.

This was a mod in Python 3.X, but isn't a showstopper: delegation-based classes can still redefine operator-overloading methods to delegate them to wrapped objects, either manually or via tools or superclasses. It's also too advanced to explore further in this tutorial, but watch for it to crop up in [Chapter 38](#), be solved in [Chapter 39](#), and make a cameo appearance as a special case in the formal inheritance definition of [Chapter 40](#).

Step 6: Using Introspection Tools

Let's make one final tweak before we throw our objects onto a database. As they are, our classes are complete and demonstrate most of the basics of OOP in Python. They still have two remaining issues we probably should iron out, though, before we go live with them:

- First, if you look at the display of the objects as they are right now, you'll notice that when you print `pat` the `Manager`, the display labels it as a `Person`. That's not technically incorrect, since `Manager` is a kind of customized and specialized `Person`. Still, it would be more accurate to display an object with the most specific (that is, *lowest*) class possible: the one an object is made from.
- Second, and perhaps more importantly, the current display format shows *only* the attributes we include in our `__repr__`, and that might not account for future goals. For example, we can't yet verify that `pat`'s job name has been set to `mgr` correctly by `Manager`'s constructor, because the

`__repr__` we coded for `Person` does not print this field. Worse, if we ever expand or otherwise change the set of attributes assigned to our objects in `__init__`, we'll have to remember to also update `__repr__` for new names to be displayed, or it will become out of sync over time.

The last point means that, yet again, we've made potential extra work for ourselves in the future by introducing *redundancy* in our code. Because any disparity in `__repr__` will be reflected in the program's output, this redundancy may be more obvious than the other forms we addressed earlier; still, avoiding extra work in the future is a generally good thing.

Special Class Attributes

We can address both issues with Python's *introspection tools*—special attributes and functions that give us access to some of the internals of objects' implementations. These tools are somewhat advanced and generally used more by people writing tools for other programmers to use than by programmers developing applications. Even so, a basic knowledge of some of these tools is useful because they allow us to write code that processes classes in generic ways. In our code, for example, there are two hooks that can help us out, both of which were introduced near the end of the preceding chapter and used in earlier examples:

- The built-in `instance.__class__` attribute provides a link from an instance to the class from which it was created. Classes in turn have a `__name__`, just like modules, and a `__bases__` sequence that provides access to superclasses. We can use these here to print the name of the class from which an instance is made rather than one we've hardcoded.
- The built-in `object.__dict__` attribute provides a dictionary with one key/value pair for every attribute attached to a namespace object (including modules, classes, and instances). Because it is a dictionary, we can fetch its keys list, index by key, iterate over its keys, and so on, to process all attributes generically. We can use this here to print every attribute in any instance, not just those we hardcode in custom displays, much as we did in [Chapter 25's](#) module tools.

We met the first of these categories in the prior chapter, but here's a quick review at Python's interactive prompt with the latest versions of our *person.py* classes ([Example 28-10](#)). Notice how we load `Person` at the interactive prompt with a `from` statement here—class names live in and are imported from modules, exactly like function names and other variables:

```
>>> from person import Person
>>> pat = Person('Pat Jones')
>>> pat                                     # Run pat's __repr__
[Person: Pat Jones $0]

>>> pat.__class__                           # Show pat's class and its name
<class 'person.Person'>
>>> pat.__class__.__name__
'Person'

>>> list(pat.__dict__.keys())                # Attributes are dict keys
['name', 'job', 'pay']

>>> for key in pat.__dict__:
    print(key, '=>', pat.__dict__[key])     # Index manually: no inheritance

name => Pat Jones
job => None
pay => 0
```

```
>>> for key in pat.__dict__:
    print(key, '=>', getattr(pat, key))
```

obj.attr, but attr is a string
Runs attr inheritance

...same as prior output...

As noted briefly in the prior chapter, some attributes accessible from an instance might not be stored in the `__dict__` dictionary if the instance's class defines a `__slots__`: an optional and relatively obscure feature of classes that stores attributes sequentially in the instance; may preclude an instance `__dict__` altogether; and which we won't study in full until [Chapter 32](#). Since slots really belong to classes instead of instances, and since they are rarely used in any event, we can reasonably ignore them here and focus on the normal `__dict__`.

As we do, though, keep in mind that some programs may need to catch exceptions for a missing `__dict__`, or use built-in functions like `hasattr`, `getattr`, and `dir` if its users might deploy slots. As you'll learn in [Chapter 32](#), the next section's code won't fail if used by a class with slots (its lack of them is enough to guarantee a `__dict__`) but slots—and other “virtual” attributes—won't be reported as instance data.

A Generic Display Tool

We can put these interfaces to work in a superclass that displays accurate class names and formats all attributes of an instance of any class. This superclass is coded in [Example 28-13](#)—it's a new, independent module named `classtools.py`. Because its `__repr__` display overload uses generic introspection tools, it will work on *any instance*, regardless of the instance's attributes set. And because this is a class, it automatically becomes a general formatting tool: thanks to inheritance, it can be mixed into *any class* that wishes to use its display format.

As an added bonus, if we ever want to *change* how instances are displayed we need only change this class—every class that inherits its `__repr__` will automatically pick up the new format when next used by a program.

Example 28-13. `classtools.py` (new)

"Assorted class utilities and tools"

```
class AttrDisplay:
    """
    Provides an inheritable display overload method that shows
    instances with their class names and a name=value pair for
    each attribute stored on the instance itself (but not attrs
    inherited from its classes). Can be mixed into any class,
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append(f'{key}={getattr(self, key)}')
        return ', '.join(attrs)

    def __repr__(self):
        return f'[{self.__class__.__name__}: {self.gatherAttrs()}]'

if __name__ == '__main__':
```

```

class TopTest(AttrDisplay):
    count = 0
    def __init__(self):
        self.attr1 = TopTest.count
        self.attr2 = TopTest.count+1
        TopTest.count += 2

class SubTest(TopTest):
    pass

X, Y = TopTest(), SubTest()      # Make two instances
print(X)                       # Show all instance attrs
print(Y)                       # Show lowest class name

```

Notice the docstrings here—because this is a general-purpose tool, we want to add some functional documentation for potential users to read. As we saw in [Chapter 15](#), docstrings can be placed at the top of simple functions and modules, and also at the start of classes and any of their methods; the `help` function and the PyDoc tool extract and display these automatically. We'll revisit docstrings for classes in [Chapter 29](#).

When run directly, this module's self-test makes two instances and prints them; the `__repr__` defined here shows the instance's class, and all its attributes' names and values, in sorted attribute name order:

```

$ python3 classtools.py
[TopTest: attr1=0, attr2=1]
[SubTest: attr1=2, attr2=3]

```

Another design note here: because this class uses `__repr__` instead of `__str__` its displays are used in all contexts, but its clients also won't have the option of providing an alternative low-level display—they can still add a `__str__`, but this applies to `print` and `str` only. In a more general tool, using `__str__` instead limits a display's scope, but leaves clients the option of adding a `__repr__` for a secondary display at interactive prompts and nested appearances. We'll follow this alternative policy when we code expanded versions of this class in [Chapter 31](#); for this demo, we'll stick with the all-inclusive `__repr__`.

Instance Versus Class Attributes

If you study the `classtools` module's self-test code long enough, you'll notice that its class displays only *instance attributes*, attached to the `self` object at the bottom of the inheritance tree; that's what `self.__dict__` contains. As an intended consequence, we don't see attributes inherited by the instance from classes above it in the tree.

For example, the attribute `TopTest.count` used as an instance counter in this file's self-test code is a *class attribute* that lives only on the class: it's assigned in the class block like methods, and referenced in methods by explicit class name—much like explicit calls to customized class methods. It's also inherited by instances, but inherited class attributes are attached to the class only, not copied down to instances. There's more on such nonmethod class attributes in the next chapter; the main point here is that `count` won't be displayed by `AttrDisplay`.

If you ever do wish to include inherited attributes too, you can climb the `__class__` link to the instance's class, use the `__dict__` there to fetch class attributes, and then iterate through the class's `__bases__` attribute to climb to even higher superclasses, repeating as necessary. If you're a fan of simple code, running a built-in `dir` call on the instance instead of using `__dict__` and climbing would have much the same effect, since `dir` results include inherited names in sorted results lists (along with built-in `__X__` methods that are easy to filter out as usual):

```
$ python3
>>> from person_10 import Person
>>> pat = Person('Pat Jones')

>>> list(pat.__dict__)
['name', 'job', 'pay']

>>> [a for a in dir(pat) if not a.startswith('__')]
['giveRaise', 'job', 'lastName', 'name', 'pay']
```

In the interest of space, we'll leave optional display of inherited class attributes with either tree climbs or `dir` as suggested experiments for now. For more hints on this front, though, watch for the *classtree.py* inheritance-tree climber we will write in [Chapter 29](#), and the *lister.py* attribute listers and climbers we'll code in [Chapter 31](#).

Name Considerations in Tool Classes

One last subtlety here: because our `AttrDisplay` class in the `classtools` module is a general tool designed to be mixed into other arbitrary classes, we have to be aware of the potential for unintended *name collisions* with client classes. As is, the code assumes that client subclasses may want to use both its `__repr__` and `gatherAttrs`, but the latter of these may be more than a subclass expects—if a subclass innocently defines a `gatherAttrs` name of its own, it will likely break our class, because the lower version in the subclass will be used instead of ours.

To see this for yourself, add a `gatherAttrs` to `TopTest` in the file's self-test code; unless the new method is identical, or intentionally customizes the original, our tool class will no longer work as planned—`self.gatherAttrs` within `AttrDisplay` searches anew from the `TopTest` instance:

```
class TopTest(AttrDisplay):
    ...
    def gatherAttrs(self):          # Replaces method in AttrDisplay!
        return 'Oops'
```

This isn't necessarily bad—sometimes we want other methods to be available to subclasses, either for direct calls or for customization this way. If we really meant to provide a `__repr__` only, though, this is less than ideal.

To minimize the chances of name collisions like this, Python programmers often prefix methods not meant for external use with a *single underscore*: `_gatherAttrs` in our case. This isn't foolproof (what if another class defines its own “private” `_gatherAttrs`, too?), but it's usually sufficient, and it's a common Python naming convention for methods internal to a class.

A better solution would be to use *two underscores* at the front of the method name only: `__gatherAttrs` for us. Python automatically expands such names to include the enclosing class's name, which makes them truly unique when looked up by the inheritance search. This is a feature usually called *pseudoprivate class attributes*, which we'll expand on in [Chapter 31](#) and deploy in an expanded version of this class there. For now, we'll make both our methods available.

Our Classes' Final Form

Now, to use the preceding section's generic display tool in our classes, all we need to do is import it from its module, mix it in by inheritance in our top-level class, and get rid of the more specific `__repr__` we coded before. The new display overload method will be inherited by instances of `Person`, as well as `Manager`; `Manager` gets `__repr__` from `Person`, which now obtains it from the `AttrDisplay` coded in

another module. [Example 28-14](#) codes the final version of our *person.py* file with these changes applied, and imports and uses the separate [Example 28-13](#).

Example 28-14. person_14.py (final)

```
"""
Record and process information about people.
Run this file directly to test its classes.
"""

from classtools import AttrDisplay          # Use generic display tool

class Person(AttrDisplay):                  # Mix in a repr at this level
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):                      # Assumes last is last
        return self.name.split()[-1]

    def giveRaise(self, percent):           # Percent must be 0..1
        self.pay = int(self.pay * (1 + percent))

class Manager(Person):
    """
    A customized Person with special requirements
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)    # Job name is implied

    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    pat = Manager('Pat Jones', 50000)
    pat.giveRaise(.10)
    print(pat.lastName())
    print(pat)
```

As this is the final revision, we've added a few *comments* here to document our work—docstrings for functional descriptions and # for smaller notes, per best-practice conventions, as well as *blank lines* between methods for readability—a generally good style choice when classes or methods grow large, which has been resisted earlier for these small classes, in part to save space and keep the code more compact.

When we run this code now, we see all the attributes of our objects, not just the ones we hardcoded in the original `__repr__`. And our final issue is resolved: because `AttrDisplay` takes class names off the `self` instance directly, each object is shown with the name of its closest (lowest) class—`pat` displays as a

Manager now, not a Person, and we can finally verify that job name is correctly filled in by the Manager constructor:

```
$ python3 person_14.py
[Person: job=None, name=Bob Smith, pay=0]
[Person: job=dev, name=Sue Jones, pay=100000]
Smith Jones
[Person: job=dev, name=Sue Jones, pay=110000]
Jones
[Manager: job=mgr, name=Pat Jones, pay=60000]
```

This is the more useful display we were after. From a larger perspective, though, our attribute display class has become a *general tool*, which we can mix into any class by inheritance to leverage the display format it defines. Further, all its clients will automatically pick up future changes in our tool. Later in the book, we'll explore even more powerful class tool concepts, such as decorators and metaclasses; along with Python's many introspection tools, they allow us to write code that augments and manages classes in structured and maintainable ways.

Step 7 (Final): Storing Objects in a Database

At this point, our work is almost complete. We now have a *two-module system* that not only implements our original design goals for representing people but also provides a general attribute display tool we can use in other programs in the future. By coding functions and classes in module files, we've ensured that they naturally support reuse. And by coding our software as classes, we've ensured that it naturally supports extension.

Although our classes work as planned, though, the objects they create are not real database records. That is, if we kill Python or our program, our instances will disappear—they're transient objects in memory and are not stored in a more permanent medium like a file, so they won't be available in future program runs. It turns out that it's easy to make instance objects more permanent, with a Python feature called *object persistence*—making objects live on after the program that creates them exits. As a final step in this tutorial, let's make our objects permanent.

Pickles and Shelves

Object persistence is implemented by three standard-library modules, installed with Python itself:

`pickle`

Serializes arbitrary Python objects to and from a string of bytes

`dbm`

Implements an access-by-key filesystem for storing strings

`shelve`

Uses the other two modules to store Python objects on a file by key

We used `pickle` and noted `shelve` briefly in [Chapter 9](#) when we studied file basics. They provide simple yet useful data-storage options. Although we can't do them complete justice in this tutorial or book, they are straightforward enough that a brief introduction should suffice to get you started.

The pickle module

The `pickle` module is a general formatting and deformatting tool: given a nearly arbitrary Python *object* in memory, it converts the object to a string of bytes, which it can use later to reconstruct the

original object in memory. The `pickle` module can handle most Python objects—including lists, dictionaries, nested combinations thereof, and class instances. The latter are especially useful things to pickle because they provide both data (attributes) and behavior (methods); the combination is roughly equivalent to “records” and “programs.”

Because `pickle` is so general, it can replace extra code you might otherwise write to create and parse custom text-file representations for your objects. By storing an object’s pickle string on a file, you effectively make it persistent: simply load and unpickle it later to re-create the original object.

The `shelve` module

Although it’s easy to use `pickle` by itself to store objects in simple flat files and load them from there later, the `shelve` module provides an extra layer of structure that allows you to store pickled objects by *key*. `shelve` translates an object to its pickled string with `pickle` and stores that string under a key in a `dbm` file; when later loading, `shelve` fetches the pickled string by key and re-creates the original object in memory with `pickle`. To your script a shelf of pickled objects looks like a *dictionary*—you index by key to fetch, assign to keys to store, and use dictionary tools such as `len`, `in`, and `dict.keys` to get information. Shelves automatically map most dictionary operations to pickled objects stored in a file.²

In fact, to your script, the main coding difference between shelves and normal dictionaries is that you must *open* shelves initially and must *close* them after making changes. The net effect is that `shelve` provides a simple database for storing and fetching native Python objects by keys, and thus makes them persistent across program runs. It does not support query tools such as SQL (but Python’s `sqlite3` standard-library module does) and does not have some advanced features found in enterprise-level databases such as true transaction processing (but other Python database tools do). It does, however, store native Python objects that may be processed with the full power of the Python language once they are fetched back by key.

Storing Objects on a `shelve` Database

Pickles and shelves are somewhat advanced topics, and we won’t go into all their details here; you can read more about them in the standard-library manuals. This is all simpler in code than narrative, though, so let’s jump right in.

First up is a new script to store objects of our classes on a shelf. Since this is a new file, we’ll need to import our classes in order to make instances to store. We used `from` to load a class earlier, but as noted in the prior chapter, there are two ways to get a class from a file (class names are variables like any other, and not at all magic in this context). As an abstract refresher:

```
import person                                # Load class with import
bob = person.Person(...)                    # Go through module name

from person import Person                    # Load class with from
bob = Person(...)                          # Use name directly
```

We’ll use `from` to load in our script, just because it’s less to type. To also keep this simple, we’ll duplicate some of the self-test lines from `person.py` that make instances of our classes, so we have something to store (we won’t fret over the test-code redundancy in this simple demo). Once we have some instances,

² Terminology note: this book now uses the noun “shelf” for the object storage managed by module `shelve`, and its plural “shelves” for more than one “shelf” managed by `shelve`. In the past, the module’s verb name “shelve” was also confusingly used as the noun—much to the chagrin of this book’s editors over the years, both electronic and human.

it's almost trivial to store them on a shelf. We simply import the `shelve` module, open a new shelf with an external filename, assign the objects to keys in the shelf, and close the shelf when we're done because we've made changes—all per [Example 28-15](#).

Example 28-15. `madeb.py` (store `Person` objects in a `shelve` database)

```
from person_14 import Person, Manager          # Load our classes
bob = Person('Bob Smith')                      # Re-create objects to be stored
sue = Person('Sue Jones', job='dev', pay=100000)
pat = Manager('Pat Jones', 50000)

import shelve
db = shelve.open('persondb')                  # Filename where objects are stored
for obj in (bob, sue, pat):                    # Use object's name attr as key
    db[obj.name] = obj                         # Store object in shelf by key
db.close()                                    # Close after making changes
```

Notice how we assign objects to the shelf using their own names as keys. This is just for convenience; in a shelf, the *key* can be any string, including one we might create to be unique using tools such as process IDs and timestamps (available in the `os` and `time` standard-library modules). The only rule is that the keys must be strings and should be unique since we can store just one object per key—though that object can be a list, dictionary, or other object containing many objects itself.

In fact, the *values* we store under keys can be Python objects of almost any sort: built-in types like strings, lists, tuples, and dictionaries, as well as user-defined class instances, and nested combinations of all of these and more. For example, the `name` and `job` attributes of our objects could be nested dictionaries and lists as in earlier incarnations in this book (though this would require a bit of redesign to the current code).

That's all there is to it—if this script has no output when run, it means it probably worked; we're not printing anything, just creating and storing objects in a file-based database:

```
$ python3 madeb.py
```

Exploring Shelves Interactively

At this point, there are one or more files in the current directory whose names all start with *persondb*. The actual files created can vary, and just as in the built-in `open` function, the filename in `shelve.open()` is relative to the current working directory (CWD) unless it includes a directory path. Wherever they are stored, these files implement a keyed-access file that contains the pickled representation of our three Python objects. Don't delete these files—they are your database and are what you'll need to copy or transfer when you back up or move your storage.

You can inspect the shelf's files either from a file explorer, console shell, or Python REPL, but they are binary hash files, and most of their content makes little sense outside the context of the `shelve` module. For example, Python's standard-library `glob` module allows us to get directory listings to verify the shelf (it's just one file on macOS, *persondb.db*), and we can open it in binary mode to explore stored bytes:

```
>>> import glob
>>> glob.glob('persondb*')
['persondb.db']
>>> print(open('persondb.db', 'rb').read())
b'\x00\x06\x15a\x00\x00\x00\x02\x00\x00\x04\xd2\x00\x00...much more omitted...
```

This content can vary, but it's nearly impossible to decipher here, and doesn't exactly qualify as a user-friendly database interface. To verify better, we can write another script, or poke around our shelf at the interactive prompt. Because shelves are Python objects containing Python objects, we can process them with normal Python syntax and development modes. Here, the REPL effectively becomes a *database client*:

```
$ python3
>>> import shelve
>>> db = shelve.open('persondb')           # Reopen the shelf
>>> len(db)                                 # Three objects stored
3
>>> list(db.keys())                         # keys is the index
['Bob Smith', 'Sue Jones', 'Pat Jones']

>>> pat = db['Pat Jones']                  # Fetch an object by key
>>> pat.lastName()                         # Runs lastName from Person
'Jones'
>>> pat                                     # Runs __repr__ from AttrDisplay
[Manager: job=mgr, name=Pat Jones, pay=50000]

>>> for key in sorted(db):                 # Iterate, sort, fetch, print
    print(key, '=>', db[key])

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Pat Jones => [Manager: job=mgr, name=Pat Jones, pay=50000]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
```

Notice that we don't have to import our `Person` or `Manager` classes here in order to load or use our stored objects. For example, we can call `pat`'s `lastName` method freely, and get its custom print display format automatically, even though we don't have the `Person` class in scope here. This works because when Python pickles a class instance, it records the instance's `self` attributes, along with the names of the class it was created from and the module where that class lives. When an instance is later fetched from the shelf and unpickled, Python automatically reimports the class by name and makes a new instance of it having the previously stored instance attributes.

The upshot of this scheme is that class instances automatically acquire all their class behavior when they are loaded in the future. We have to import our classes only to make *new* instances, not to process existing ones. Although a deliberate feature, this scheme has somewhat mixed consequences:

- The *downside* is that classes and their module's files must be *importable* when an instance is later loaded. That is, picklable classes must be coded at the top level of a module file that is accessible from a directory listed on the `sys.path` module search path (and shouldn't live in the topmost script files' module `__main__` unless they're always in that module when used). Because of this external file requirement, some programs pickle simpler objects such as dictionaries or lists, especially if they are to be transferred across networks.
- The *upside* is that changes in a class's source code file are automatically picked up when instances of the class are loaded again. There is often no need to update stored objects themselves since updating their class's code changes their behavior.

Shelves have other well-known limitations covered in Python's library manual. For simple object storage, though, shelves and pickles are easy-to-use tools for personal databases, program configurations, and more.

Updating Objects on a Shelf

One last script: let's write a program that updates an instance (record) each time it runs, to show that our objects really are *persistent*—that their current values are available every time a Python program runs. The file coded in [Example 28-16](#) prints the database and gives a raise to one of our stored objects on each run. If you trace through what's going on here, you'll notice that we're getting a lot of utility “for free”—printing our objects automatically employs the general `__repr__` overloading method, and we give raises by calling the `giveRaise` method we wrote earlier. This all just works for objects based on OOP's inheritance model, even when they live in a file.

Example 28-16. updatedb.py (modify Person object in a shelve database)

```
import shelve
db = shelve.open('persondb')           # Reopen shelf with same filename

for key in sorted(db):                 # Iterate to display database objects
    print(key, '\t=>', db[key])         # Prints with custom format

sue = db['Sue Jones']                  # Index by key to fetch
sue.giveRaise(.10)                     # Update in memory using class's method
db['Sue Jones'] = sue                  # Assign to key to update in shelf
db.close()                             # Close after making changes
```

Because this script prints the database when it starts up, we have to run it at least twice to see our objects change. Here it is in action, displaying all records and increasing `sue`'s pay each time it is run (it's a pretty good script for `sue`; something to schedule to run regularly as a cron job perhaps?):

```
$ python3 updatedb.py
Bob Smith    => [Person: job=None, name=Bob Smith, pay=0]
Pat Jones    => [Manager: job=mgr, name=Pat Jones, pay=50000]
Sue Jones    => [Person: job=dev, name=Sue Jones, pay=100000]

$ python3 updatedb.py
Bob Smith    => [Person: job=None, name=Bob Smith, pay=0]
Pat Jones    => [Manager: job=mgr, name=Pat Jones, pay=50000]
Sue Jones    => [Person: job=dev, name=Sue Jones, pay=110000]

$ python3 updatedb.py
Bob Smith    => [Person: job=None, name=Bob Smith, pay=0]
Pat Jones    => [Manager: job=mgr, name=Pat Jones, pay=50000]
Sue Jones    => [Person: job=dev, name=Sue Jones, pay=121000]
```

Again, what we see here is a product of the `shelve` and `pickle` tools we get from Python, and of the behavior we coded in our classes ourselves. And once again, we can verify our script's work at the interactive prompt—the shelf's equivalent of a database client:

```
$ python3
>>> import shelve
>>> db = shelve.open('persondb')      # Reopen database
>>> rec = db['Sue Jones']              # Fetch object by key
>>> rec
[Person: job=dev, name=Sue Jones, pay=133100]
>>> rec.lastName(), rec.pay
('Jones', 133100)
```

For another example of object persistence in this book, watch for the sidebar “[Why You Will Care: Classes and Persistence](#)” on page 786. It stores a somewhat larger composite object in a flat file with `pickle` instead of `shelve`, but the effect is similar. For more details and examples of pickles, see also [Chapter 9](#) (file basics) and [Chapter 37](#) (string tools and Unicode), as well as Python’s manuals.

Future Directions

And that’s a wrap for this chapter’s tutorial. At this point, you’ve seen all the basics of Python’s OOP machinery in action, and you’ve learned ways to avoid redundancy and its associated maintenance issues in your code. You’ve built full-featured classes that do real work. As an added bonus, you’ve made them real database records by storing them in a Python shelf, so their information lives on persistently.

There is much more we could explore here, of course. For example, we could extend our classes to make them more realistic, add new kinds of behavior to them, and so on. Giving a raise, for instance, should in practice verify that pay increase rates are between zero and one—an extension we’ll add when we meet decorators later in this book. You might also mutate this example into a personal contacts database, by changing the state information stored on objects, as well as the classes’ methods used to process it. We’ll leave this a suggested exercise open to your imagination.

We could also expand our scope to use tools that either come with Python or are freely available in the open source world. For instance, GUIs, websites, or apps would make our database more accessible to nonprogrammers, and other database interfaces like `sqlite3` and content-representation schemes like JSON offer additional possibilities.

While this chapter has hopefully sparked your interest for future exploration, such topics are of course beyond the scope of this tutorial and this book at large. If you want to explore any of them on your own, see the web, Python’s standard-library manuals, and follow-up application texts. First, though, let’s return to class fundamentals and finish up the rest of the Python core-language story.

Chapter Summary

In this chapter, we explored all the fundamentals of Python classes and OOP in action, by building upon a simple but real example, step by step. We added constructors, methods, operator overloading, customization with subclasses, and introspection-based tools, and we met concepts such as composition, delegation, and polymorphism along the way.

In the end, we took objects created by our classes and made them persistent by storing them on a `shelve` object database—a simple system for saving and retrieving native Python objects by key. While exploring class basics, we also encountered multiple ways to factor our code to reduce redundancy and minimize future maintenance costs.

In the next chapters of this part of the book, we’ll resume our study of the details behind Python’s class model and investigate its application to some of the design concepts used to combine classes in larger programs. Before we move ahead, though, let’s work through this chapter’s quiz to review what we covered here. Since we’ve already done a lot of hands-on work in this chapter, we’ll close with a set of mostly theory-oriented questions designed to make you trace through some of the code and ponder some of the bigger ideas behind it.

Test Your Knowledge: Quiz

1. When we fetch a `Manager` object from the shelf and print it, where does the display format logic come from?
2. When we fetch a `Person` object from a shelf without importing its module, how does the object know that it has a `giveRaise` method that we can call?
3. Why is it so important to move processing into methods, instead of hardcoding it outside the class?
4. Why is it better to customize by subclassing rather than copying the original and modifying?
5. Why is it better to call back to a superclass method to run default actions, instead of copying and modifying its code in a subclass?
6. Why is it better to use tools like `__dict__` that allow objects to be processed generically than to write more custom code for each type of class?
7. In general terms, when might you choose to use object embedding and composition instead of inheritance?
8. What would you have to change if the objects coded in this chapter used a dictionary for names and a list for jobs, as in similar examples earlier in this book?
9. How might you modify the classes in this chapter to implement a personal contacts database in Python?

Test Your Knowledge: Answers

1. In the final version of our classes, `Manager` ultimately inherits its `__repr__` printing method from `AttrDisplay` in the separate `classtools` module, and two levels up in the class tree. `Manager` doesn't have one itself, so the inheritance search climbs to its `Person` superclass; because there is no `__repr__` there either, the search climbs higher and finds it in `AttrDisplay`. The class names listed in parentheses in a class statement's header line provide the links to higher superclasses.
2. Shelves (really, the `pickle` module they use) automatically relink an instance to the class it was created from when that instance is later loaded back into memory. Python reimports the class from its module internally and creates a new instance with the previously stored attributes. This way, loaded instances automatically obtain all their original methods (like `lastName`, `giveRaise`, and `__repr__`), even if we have not imported the instance's class into the loading scope.
3. It's important to move processing into methods so that there is only one copy to change in the future, and so that the methods can be run on any instance. This is Python's notion of *encapsulation*—wrapping up logic behind interfaces, to better support future code maintenance. If you don't do so, you create code redundancy that can multiply your work effort as the code evolves in the future.
4. Customizing with subclasses reduces development effort. In OOP, we code by *customizing* what has already been done, rather than copying or changing existing code. This is the real “big idea” in OOP—because we can easily extend our prior work by coding new subclasses, we can leverage what we've already done. This is much better than either starting from scratch each time, or introducing multiple redundant copies of code that may all have to be updated in the future.

5. Copying and modifying code *doubles* your potential work effort in the future, regardless of the context. If a subclass method needs to perform default actions coded in a superclass method, it's much better to call back to the original through the superclass's name (or the `super` built-in) than to copy its code. This also holds true for superclass constructors. Again, copying code creates redundancy, which is a major issue as code evolves.
6. Generic tools can avoid hardcoded solutions that must be kept in sync with the rest of the class as it evolves over time. A generic `__repr__` print method, for example, need not be updated each time a new attribute is added to instances in an `__init__` constructor. In addition, a generic `print` method inherited by all classes appears and need be modified in only one place—changes in the generic version are picked up by all classes that inherit from the generic class. Again, eliminating code *redundancy* cuts future development effort; that's one of the primary assets classes bring to the table.
7. Inheritance is best at coding extensions based on direct customization (like our `Manager` specialization of `Person`). Composition is well suited to scenarios where multiple objects are aggregated into a whole and directed by a controller-layer class. Inheritance passes calls *up* to reuse, and composition passes *down* to delegate. Inheritance and composition are not mutually exclusive; often, the objects embedded in a controller are themselves customizations based upon inheritance.
8. Not much since this was really a first-cut prototype, but the `lastName` method would need to be updated for the new name format; the `Person` constructor would have to change the job default to an empty list; and the `Manager` class would probably need to pass along a job list in its constructor instead of a single string (self-test code would change as well, of course). The good news is that these changes would need to be made in just one place—in our classes, where such details are encapsulated. Apart from their object-construction calls, the database scripts should work as is, as shelves support arbitrarily nested data.
9. The classes in this chapter could be used as boilerplate “template” code to implement a variety of types of databases. Essentially, you can repurpose them by modifying the constructors to record different attributes and providing whatever methods are appropriate for the target application. For instance, you might use attributes such as `name`, `address`, `birthday`, `phone`, `email`, and so on for a contacts database, and methods appropriate for this purpose. A method named `sendmail`, for example, might use Python's standard-library `smtplib` module to send an email to one of the contacts automatically when run (see Python's manuals or application-level books for more details on such tools). The `AttrDisplay` tool we wrote here could be used verbatim to print your objects because it is intentionally generic. Most of the `shelve` database code here can be used to store your objects, too, with minor changes.

Class Coding Details

If you haven't quite grasped all of Python OOP yet, don't worry—now that we've taken a first pass, we're going to dig a bit deeper and study the concepts introduced earlier in further detail. In this and the following chapter, we'll take another look at class mechanics. Here, we'll study classes, methods, and inheritance, formalizing and expanding on some of the coding ideas introduced in [Chapter 27](#) and demoed in [Chapter 28](#). Because the class is our last namespace tool, we'll summarize Python's namespace and scope concepts as well.

If you've been reading linearly, some of this chapter will be partly review and summary of topics introduced in the preceding chapter's case study, revisited here by language topics with self-contained examples that may help readers new to OOP. While you may be tempted to skip some material here, it includes extra details worth a browse and unveils more subtleties in Python's class model along the way.

The next chapter continues this in-depth second pass over class mechanics by covering one specific aspect: operator overloading. First, though, let's fill in more of the Python OOP picture.

The class Statement

Although the Python `class` statement may seem similar to tools in other OOP languages on the surface, on closer inspection, it is quite different from what some programmers may be used to.

For example, as in C++, the `class` statement is Python's main OOP tool, but unlike in C++, Python's `class` is not a declaration. Like a `def`, a `class` statement is an object builder and an implicit assignment—when run, it generates a class object and stores a reference to it in the name used in the header. Also like a `def`, a `class` statement is true executable code—your class doesn't exist until Python reaches and runs the `class` statement that defines it. This typically occurs while importing the module it is coded in, but not before.

General Syntax and Usage

As we've seen, `class` is a compound statement with a body of statements typically indented under the header. In the header, superclasses are listed in parentheses after the class name, separated by commas. Listing more than one superclass leads to multiple inheritance, which we'll discuss more formally in [Chapter 31](#) (in brief, the left-to-right order of superclasses in parentheses gives the search order). Here is the statement's general form and usage:

```

class name(superclass,...):           # Assign to name
    attr = value                      # Shared class data
    def method(self,...):            # Methods
        self.attr = value           # Per-instance data

x = name(...)                         # Make an instance
x.method(...)                        # Call a method

```

Within the `class` statement, any *assignments* generate class attributes (both data items and callable functions known as *methods*); specially named methods implement built-in *operations* (e.g., a function named `__init__` is run at instance-object construction time if defined); and calling the class after its `class` has run makes *instances* of it.

Example: Class Attributes

As we’ve also seen, classes are mostly just *namespaces*—tools for defining names (i.e., attributes) that export data and logic to clients. Just as in a module file, the statements nested in a `class` statement body create its namespace and attributes. When Python reaches and runs a `class` statement, it runs all the statements nested in its body, from top to bottom. Assignments that happen during this process create names in the class’s local scope, which become attributes in the associated class object. Because of this, classes resemble both *modules* and *functions*:

- Like functions, `class` statements are local scopes where names created by nested assignments live.
- Like modules, names assigned in a `class` statement become attributes in a class object.

The main distinction for classes is that their namespaces are also the basis of *inheritance* in Python: referenced attributes that are not found in a class or instance object may be fetched from other classes.

Because `class` is a compound statement, any sort of statement can be nested inside its body—`print`, assignments, `if`, `def`, and so on. All the statements inside the `class` statement run when the `class` statement itself runs (not when the class is later *called* to make an instance). Typically, assignment statements inside the `class` statement make data attributes and nested `defs` make method attributes. In general, though, any type of name assignment at the top level of a `class` statement creates a same-named attribute in the resulting class object.

For example, assignments of simple nonfunction objects to class attributes produce *data attributes* shared by all instances. In the REPL of your choosing:

```

>>> class SharedData:
    attr = 16                      # Generates a class data attribute

>>> x = SharedData()              # Make two instances
>>> y = SharedData()
>>> x.attr, y.attr                # They inherit and share 'attr' (a.k.a. SharedData.attr)
(16, 16)

```

Here, because the name `attr` is assigned at the top level of a `class` statement, it is attached to the *class*—which means it will be shared by all instances via the usual inheritance search from instance to class. We can change it by going through the class name, and we can refer to it through either instances or the class:

```

>>> SharedData.attr = 32
>>> x.attr, y.attr, SharedData.attr
(32, 32, 32)

```

Such class attributes can be used to manage information that spans all the instances—a counter of the number of instances generated, for example (an idea we’ll expand on by example in [Chapter 32](#)). Now, watch what happens if we assign the name `attr` through an instance instead of the class:

```
>>> x.attr = 64
>>> x.attr, y.attr, SharedData.attr
(64, 32, 32)
```

Assignments to instance attributes create or change the names in the *instance*, not the shared class. More generally, inheritance searches occur only on attribute *references*, not on attribute *assignments*: assigning to an object’s attribute always changes that object and no other (subject to the note ahead). For example, `y.attr` is still looked up in the class by inheritance, but the assignment to `x.attr` attaches a name to `x` itself and so replaces the version in the class.

Readers who’ve done OOP before may recognize class attributes like `SharedData.attr` as similar to other languages’ “static” data members—values that are stored in the class, independent of instances. In Python, it’s nothing special: all class attributes are just names assigned in the `class` statement, whether they happen to reference functions or something else. When they are functions (a.k.a. methods), they simply receive an instance when called through one.

Here’s a more comprehensive example of this behavior that stores the same name in two places. Suppose we run the following class in a REPL:

```
>>> class MixedNames:
    data = 'text'
    def __init__(self, value):
        self.data = value
    def display(self):
        print(self.data, MixedNames.data)
```

Define class
Assign class attr
Assign method name
Assign instance attr
Instance attr, class attr

This class contains two defs, which assign class attributes to method functions. It also contains a top-level = assignment statement; because this assignment assigns the name `data` inside the `class`, it lives in the class’s local scope and becomes an attribute of the class object. Like all class attributes, this `data` is inherited and shared by all instances of the class that don’t have `data` attributes of their own.

When we make instances of this class, though, the name `data` is *also* attached to those instances by the assignment to `self.data` in the `__init__` method run automatically at instance-construction time:

```
>>> x = MixedNames(1)
>>> y = MixedNames(2)
>>> x.display(); y.display()
```

Make two instance objects
Each has its own data
self.data differs, MixedNames.data is the same
1 text
2 text

The net result is that `data` lives in *two* places: in the instance objects (created by the `self.data` assignment in `__init__`) and in the class from which they inherit names (created by the `data` assignment in the class). The class’s `display` method prints both versions by first qualifying the `self` instance and then the class.

By using these techniques to store attributes in different objects, we determine their scope of visibility. When attached to classes, names are shared. When attached to instances, names record per-instance data, not shared behavior or data. Although inheritance searches look up names for us, we can always get to an attribute anywhere in a tree by accessing the desired object directly. The object from which an attribute is requested focuses and limits search.

In the preceding example, for instance, specifying `x.data` or `self.data` will return an instance name, which normally hides the same name in the class. However, `MixedNames.data` grabs the class’s version

of the name explicitly. The next section describes another common role for such through-the-class coding patterns and explains more about the way we deployed class-level fetches in the prior chapter.



Assignment-rule exceptions: Assigning to an object’s attribute always changes only that object—*unless*, that is, the object inherits from a class that has redefined attribute assignment to do something unique with the `__setattr__` operator-overloading method (discussed in [Chapter 30](#)) or uses advanced attribute-management tools such as *properties* and *descriptors* (discussed in [Chapters 32](#) and [38](#)). Much of this chapter presents the normal case, which suffices at this point in the book and for most Python code. As you’ll see later, though, Python classes are, well, richly endowed with hooks that allow programs to deviate from the norm—and render simple rules fanciful.

Methods

Because you already know about functions, you also know about methods in classes. As you’ve learned, methods are just function objects created by `def` statements nested in a `class` statement’s body. From an abstract perspective, methods provide behavior for instance objects to inherit. From a programming perspective, methods work in exactly the same way as simple functions, with one crucial exception: a method’s first argument receives the instance object that is the implied subject of the method call.

By way of review from the last chapter, a method call made through an instance like this:

```
instance.method(args...)
```

is automatically translated into a call of method function in a class like this:

```
class.method(instance, args...)
```

where Python determines the class to use by locating the method name using the inheritance search procedure. In fact, both call forms are valid in Python: in the second, the class name narrows the method search, and the instance is provided explicitly, but the net result is the same for the same method.

Besides the inheritance of method names, the special first argument is the only real magic behind method calls. In a class’s method, the first argument is usually called `self` by convention (technically, only its position is significant, not its name). This argument provides methods with a hook back to the instance that is the subject of the call—because classes generate many instance objects, they use `self` to manage per-instance data.

In Python, `self` is always explicit in your code: methods must always both list and use `self` to fetch or change attributes of the instance being processed by the current method call. This is by design—the presence of this name makes it obvious that you are using instance attribute names in your script, not names in the local or global scope.

Method Example

To solidify these concepts, let’s turn to an example. Define the following class by running its code in a REPL:

```
>>> class NextClass:                # Define class
    def printer(self, text):         # Define method
        self.message = text         # Change instance
        print(self.message)         # Access instance
```

The name `printer` references a normal function object; because it's assigned in the `class` statement's scope, it becomes a class-object attribute and is inherited by every instance made from the class—and earns the title *method*. Normally, because methods like `printer` are designed to process instances, we call them through instances:

```
>>> x = NextClass()           # Make instance
>>> x.printer('instance call') # Call its method
instance call
>>> x.message                 # Instance changed
'instance call'
```

When we call the method by qualifying an instance like this, `printer` is first located by inheritance, and then its `self` argument is automatically assigned the instance object (`x`); the text argument gets the string passed at the call (`'instance call'`). Notice that because Python automatically passes the first argument to `self` for us, we can (and must) pass in just one argument. Inside `printer`, the name `self` is used to access or set per-instance data because it refers back to the instance currently being processed.

As we've seen, though, methods may be called in one of two ways—through an *instance* or through the *class* itself. For example, we can also call `printer` by going through the class name, provided we pass an instance to the `self` argument explicitly:

```
>>> NextClass.printer(x, 'class call') # Direct class call
class call
>>> x.message                          # Instance changed again
'class call'
```

Calls routed through the instance and the class have the exact same effect—as long as we pass the same instance object ourselves in the class form. In fact, you get an error message if you try to call our method without any instance:

```
>>> NextClass.printer('bad call')
TypeError: NextClass.printer() missing 1 required positional argument: 'text'
```

Really, class methods are just *functions* assigned to class attributes, some of which happen to expect an instance that Python provides automatically *only* when methods are called through an instance. Moreover, calling a method through a class this way uses the same pattern we coded previously to fetch *non-function* class attributes. This same expression, `class.attribute`, works the same, whether the result is a callable object or not. It's a general tool.

Other Method-Call Possibilities

This pattern of calling methods through a class is the general basis of *extending*—instead of completely replacing—inherited method behavior. It requires an explicit instance to be passed because all methods do by default. Technically, this is because methods called through instances are *instance methods* in the absence of any special code.

In [Chapter 32](#), we'll also study a less common option, *static methods*, that allows us to code methods that do not expect instance objects in their first arguments, even when called through an instance. Such methods can act like simple instanceless functions, with names that are local to the classes in which they are coded, and may be used to manage class data. A related concept we'll explore in the same chapter, *class methods* receive a class when called instead of an instance and can be used to manage per-class data, and are implied in metaclasses—yet another topic we'll reach later.

These are all advanced, optional, and atypical extensions, though. Normally, an instance must always be passed to a method—whether automatically when it is called through an instance, or manually when you call through a class.

Inheritance

Of course, the whole point of the namespace created by the `class` statement is to support name inheritance. This section expands on some of the mechanisms and roles of attribute inheritance in Python.

As we've seen, in Python, inheritance happens when an object is qualified, and it involves searching an attribute definition tree—one or more namespaces. Every time you use an expression of the form `object.attr` where `object` is an instance or class object, Python searches the namespace tree from bottom to top, beginning with `object`, and looking for the first `attr` it can find. This also happens for references to `self` attributes in your methods. Because lower definitions in the tree override higher ones, inheritance forms the basis of specialization.

Attribute Tree Construction

Figure 29-1 summarizes the way namespace trees are constructed and populated with names. Generally:

- Instance attributes are generated by assignments to `self` attributes in methods.
- Class attributes are created by statements (assignments) nested in `class` statements.
- Superclass links are made by listing classes in parentheses in a `class` statement header.

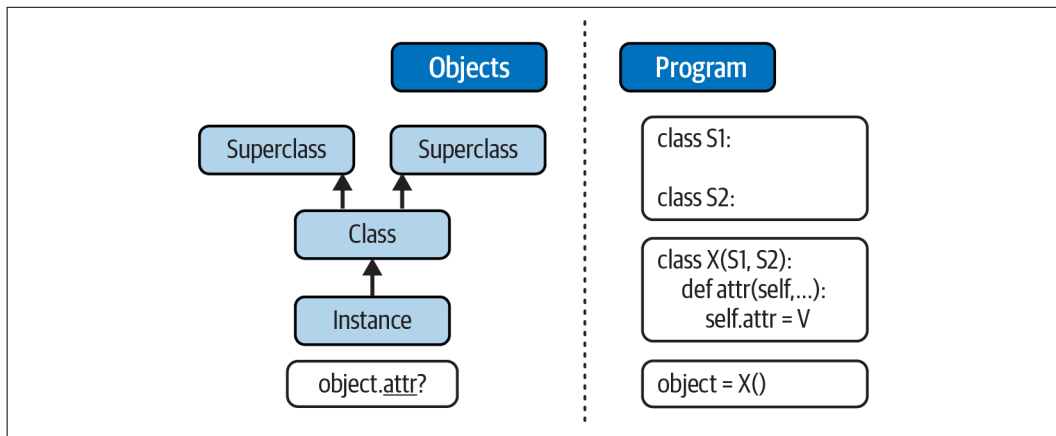


Figure 29-1. Program code creates a tree of objects searched by attribute inheritance

The net result is a tree of attribute namespaces that leads from an instance to the class it was generated from to all the superclasses listed in the `class` header. Python searches upward in this tree—from instances to superclasses, and left to right through multiple superclasses—each time you fetch an attribute name from an instance object.

Inheritance Fine Print

Technically speaking, the preceding description isn't complete because we can also create instance and class attributes by assigning them to objects outside of `class` statements. In Python, all attributes are always accessible by default, and *privacy* is an add-on (we'll talk about attribute privacy in [Chapter 30](#) when we study `__setattr__`, in [Chapter 31](#) when we meet `__X` names, and again in [Chapter 39](#) when we implement it with a class decorator). Even so, changes outside of a `class` are uncommon and error-prone: classes work best when they manage their instances.

Also technically speaking, as hinted in [Chapter 27](#), the full inheritance story grows more convoluted when advanced topics we haven't yet met are added to the mix. *Metaclasses*, diamond-pattern *MROs*, and *descriptors*, for example, may all play a role in some programs. Because of this, we'll begin formalizing the inheritance algorithm in [Chapter 31](#) but won't finish it until [Chapter 40](#). In the vast majority of Python code, though, inheritance is a simple way to redefine, and hence customize, behavior coded in classes—as the next section demos.

Specializing Inherited Methods

The tree-searching model of inheritance just described turns out to be a great way to specialize systems. Because inheritance finds names in subclasses before it checks superclasses, subclasses can replace default behavior by redefining their superclasses' attributes. In fact, you can build entire systems as hierarchies of classes, which you extend by adding new external subclasses rather than copying existing logic or changing it in place.

The idea of redefining inherited names leads to a variety of specialization techniques. For instance, subclasses may *replace* inherited attributes completely, *provide* attributes that a superclass expects to find, and *extend* superclass methods by calling back to the superclass from an overridden method. We've already seen some of these patterns in action; here's a self-contained example of extension at work:

```
>>> class Super:
    def method(self):
        print('in Super.method')

>>> class Sub(Super):
    def method(self):
        print('starting Sub.method')
        Super.method(self)
        print('ending Sub.method')
```

Direct superclass method calls are the crux of the matter here. The `Sub` class replaces `Super`'s `method` function with its own specialized version, but within the replacement, `Sub` calls back to the version exported by `Super` to carry out the default behavior. In other words, `Sub.method` just extends `Super.method`'s behavior rather than replacing it completely:

```
>>> x = Super()
>>> x.method()
in Super.method

>>> x = Sub()
>>> x.method()
starting Sub.method
in Super.method
ending Sub.method
```

Perhaps the most common places that superclass-method calls show up are in constructors. The `__init__` method, like all attributes, is looked up by inheritance. This means that at construction time, Python locates and calls just *one* `__init__`, not one in every superclass. If subclass constructors need to ensure that superclass construction-time logic runs too, they must call the superclass's `__init__` method explicitly. Calling it through the *class name* leverages the same general coding pattern we've been using in multiple roles:

```
>>> class Super:
    def __init__(self, x):
        print('default code')

>>> class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x)          # Run superclass __init__
        print('custom code')            # Do my extra init actions

>>> I = Sub(1, 2)
default code
custom code
```

This is one of the few contexts in which your code is likely to call an operator-overloading method directly. Naturally, you should call the superclass constructor this way only if you really *want* it to run—without the call, the subclass replaces it completely. For a more realistic illustration of this technique in action, see the `Manager` class example in the prior chapter's tutorial.

On a related note, readers with prior OOP experience may also be interested to know that redefining the constructor with differing argument lists in the *same* class means that only the *last* is used—later defs simply reassign the method name in Python. Starred arguments can be used in this role, but rarely are. We'll explore this phenomenon more fully in [Chapter 31](#)'s coverage of polymorphism (short story: it's about interfaces, not call signatures).



The super reminder: Per the sidebar “[The super Alternative](#)” on [page 691](#), Python also has a `super` built-in function that allows calling back to a superclass's methods more generically, but we're deferring its coverage until [Chapter 32](#) due to its downsides and complexities. As a preview, though, the prior section's first of the following can also be coded as the second—which essentially automates the `self` argument via deep magic beyond our scope here (note its lowercase):

```
Super.method(self)          # Explicit, general tool
super().method()            # Implicit, special case
```

Likewise, the same equivalence goes for the constructor calls of this section:

```
Super.__init__(self, x)      # Explicit fundamental
super().__init__(x)          # Implicit alternative
```

Per the aforementioned sidebar, though, `super` has well-known trade-offs in basic usage and an esoteric advanced use case that requires universal deployment to be most effective. Because of such issues, this book prefers to call superclasses by explicit name instead of `super`. If you're new to Python, consider following the same policy, especially for your first pass over OOP. Learn the simple and general now so you can weigh it against the complicated and narrow later.

Class Interface Techniques

Broadly speaking, the prior section's *extension* is only one way to interface with a superclass. The file listed in [Example 29-1](#), *specialize.py*, defines multiple classes that illustrate a variety of common techniques:

Super

Defines a method function and a delegate that expects an action in a subclass

Inheritor

Doesn't provide any new names, so it gets everything defined in Super

Replacer

Overrides Super's method with a version of its own

Extender

Customizes Super's method by overriding and calling back to run the default

Provider

Implements the action method expected by Super's delegate method

Study each of these subclasses to get a feel for the various ways they customize their common superclass.

Example 29-1. *specialize.py*

```
class Super:
    def method(self):
        print('in Super.method')           # Default behavior
    def delegate(self):
        self.action()                     # Expected to be defined

class Inheritor(Super):                  # Inherit method verbatim
    pass

class Replacer(Super):                  # Replace method completely
    def method(self):
        print('in Replacer.method')

class Extender(Super):                  # Extend method behavior
    def method(self):
        print('starting Extender.method')
        Super.method(self)               # Or: super().method()
        print('ending Extender.method')

class Provider(Super):                 # Fill in a required method
    def action(self):
        print('in Provider.action')

if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender):
        print('\n' + klass.__name__ + '...')
        klass().method()

    print('\nProvider...')
    x = Provider()
    x.delegate()
```

Two things are worth pointing out here. First, notice how the self-test code at the end of this example creates instances of three different classes in a `for` loop. Because classes, like functions, are *first-class objects*, you can store them in a tuple and create instances generically with no extra syntax. Second, classes also have a built-in `__name__` attribute, like modules; it's preset to a string containing the name in the class header. Here's what happens when we run the file:

```
$ python3 specialize.py
```

```
Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

Trace through the code to see how each of these outputs is produced.

Abstract Superclasses

Of the prior example's classes, `Provider` may be one of the most crucial to understand. When we call the `delegate` method through a `Provider` instance, *two* independent inheritance searches occur:

1. On the initial `x.delegate` call, Python finds the `delegate` method in `Super` by searching the `Provider` instance and above. The instance `x` is passed into the method's `self` argument as usual.
2. Inside the `Super.delegate` method, `self.action` invokes a new, independent inheritance search of `self` and above. Because `self` references a `Provider` instance, the `action` method is located in the `Provider` subclass.

This “filling in the blanks” sort of coding structure is typical of OOP frameworks. In a more realistic context, the method filled in this way might handle an event in a GUI, provide data to be rendered as part of a web page, process a tag's text in an XML file, and so on—your subclass provides specific actions, but the framework handles the rest of the overall job and runs your actions when needed.

At least in terms of the `delegate` method, the superclass in this example is what is sometimes called an *abstract superclass*—a class that expects parts of its behavior to be provided by its subclasses. If an expected method is not defined in a subclass, Python raises an `undefined name` exception when the inheritance search fails.

Class coders sometimes make such subclass requirements more obvious with `assert` statements or by raising the built-in `NotImplementedError` exception with `raise` statements. We'll study statements that may trigger exceptions in depth in the next part of this book; as a quick preview, here's the `assert` scheme in `action`:

```
>>> class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!'      # Error if called

>>> X = Super()
>>> X.delegate()
AssertionError: action must be defined!
```

We’ll study `assert` in Chapters 33 and 34; in short, if its first expression evaluates to false, it raises an exception with the provided error message. Here, the expression is always false so as to trigger an error message if a method is not redefined, and inheritance locates the stub version here. Alternatively, some classes simply raise a `NotImplementedError` exception directly in such method stubs to signal the mistake:

```
>>> class Super:
    def delegate(self):
        self.action()
    def action(self):
        raise NotImplementedError('action must be defined!')

>>> X = Super()
>>> X.delegate()
NotImplementedError: action must be defined!
```

For instances of *subclasses*, we still get the exception unless the subclass provides the expected method to replace the default in the superclass:

```
>>> class Sub(Super): pass
>>> X = Sub()
>>> X.delegate()
NotImplementedError: action must be defined!

>>> class Sub(Super):
    def action(self): print('okay')

>>> X = Sub()
>>> X.delegate()
okay
```

For a somewhat more realistic example of this section’s concepts in action, see the “Zoo animal hierarchy” exercise (Exercise 8) in “[Test Your Knowledge: Part VI Exercises](#)” on page 866 and its solution in [Appendix B](#). Such taxonomies are a traditional way to introduce OOP, but they’re a bit removed from most developers’ job descriptions (with apologies to any readers who happen to work at the zoo).

Preview: Abstract superclasses with library tools

The preceding abstract superclasses (a.k.a. “abstract base classes”), which require methods to be filled in by subclasses, may also be implemented with special class syntax and a library module. This is coded with a keyword argument in a `class` header, along with special `@` decorator syntax, both of which we’ll study later in this book. While necessarily a preview in part, here is the special syntax equivalent of the preceding example:

```
>>> from abc import ABCMeta, abstractmethod

>>> class Super(metaclass=ABCMeta):
    def delegate(self):
        self.action()
    @abstractmethod
    def action(self):
        pass
```

The net effect more rigidly prevents instance *creation* unless the method is defined lower in the class tree:

```
>>> X = Super()
TypeError: Can't instantiate abstract class Super without an implementation
for abstract method 'action'

>>> class Sub(Super): pass
>>> X = Sub()
TypeError: Can't instantiate abstract class Sub without an implementation
for abstract method 'action'

>>> class Sub(Super):
    def action(self): print('okay')

>>> X = Sub()
>>> X.delegate()
okay
```

Coded this way, a class with an abstract method cannot be instantiated (that is, we cannot create an instance by calling it) unless all of its abstract methods have been defined in subclasses. Although this requires more code and extra knowledge, the potential advantage of this approach is that errors for missing methods are issued when we attempt to *make* an instance of the class, not later when we try to *call* a missing method. This scheme may also be used to define an expected interface, automatically verified in client classes.

Unfortunately, this scheme also relies on two advanced language tools we have not mastered yet—*function decorators*, introduced in [Chapter 32](#) and covered in depth in [Chapter 39](#), as well as *metaclass declarations*, mentioned in [Chapter 32](#) and covered in [Chapter 40](#)—so we will postpone other facets of this option here. See Python’s standard manuals for more on this, as well as precoded abstract super-classes Python provides.

Namespaces: The Conclusion

Now that we’ve examined class and instance objects, the Python namespace story is complete. For reference, this section summarizes all the rules used to resolve names and extends them to classes. The first things you need to remember are that qualified and unqualified names are treated differently, and that some scopes serve to initialize object namespaces:

- Unqualified names (e.g., `X`) deal with scopes.
- Qualified attribute names (e.g., `object.X`) use object namespaces.
- Some scopes initialize object namespaces (for modules and classes).

These concepts sometimes interact—in `object.X`, for example, `object` is first looked up per scopes, and then `X` is looked up in the located object. Since scopes and namespaces are essential to understanding Python code, let’s flesh out the rules in more detail.

Simple Names: Global Unless Assigned

As we've seen, unqualified simple names follow the *LEGB* lexical scoping rule outlined when we explored functions in [Chapter 17](#):

Assignment ($X = \text{value}$)

Makes names local by default: creates or changes the name X in the current *local* scope, unless declared *global* or *nonlocal* in that scope. These declarations work in both *def* for functions and *class* for classes.

Reference (X)

Looks for the name X in the current local scope (L), then any and all enclosing *functions* from inner to outer (E), then the current global-scope *module* (G), then the *built-ins* module (B)—per the *LEGB* rule. Notably absent here, enclosing *classes* are not searched; class names are referenced as object attributes instead.

Also per [Chapter 17](#), some special-case constructs localize names further (e.g., variables in some comprehensions and some *try* statement clauses), and nested class scopes currently have some peculiarities that reflect longstanding bug reports and are too obscure to merit coverage here. The vast majority of names, however, follow the *LEGB* rule.

New here, the *class* statement allows *global* and *nonlocal* to modify assignment rules the same as *def*, though we have to *nest* it to see how the latter of these come online:

```
>>> gvar = 111
>>> class C:
    global gvar          # Change name gvar in enclosing module
    gvar = 222           # Else it would be class attribute C.gvar

>>> gvar
222

>>> def outer():
    nvar = 111
    class C:
        nonlocal nvar    # Change name nvar in enclosing function
        nvar = 222       # Else it would be class attribute C.nvar
        print(nvar)

>>> outer()
222
```

Though rare, namespace declarations in *class* map assignments to outer scopes, instead of making class attributes—the same way they prevent assignments from making local variables in functions. There's more on how nested classes interact with scopes in default cases later in this section.

Attribute Names: Object Namespaces

We've also seen that qualified attribute names refer to attributes of specific objects and obey the rules for modules and classes. For class and instance objects, the reference rules are augmented to include the inheritance search procedure:

Assignment (*object.X* = *value*)

Creates or alters the attribute name X in the namespace of the *object* being qualified, and none other. Inheritance-tree climbing happens only on attribute reference, not on attribute assignment.

Reference (object.X)

For class-based objects, searches for the attribute name *X* in *object*, then in all accessible classes above it, using the inheritance search procedure. For nonclass objects such as modules, fetches *X* from *object* directly.

As noted earlier, the preceding captures the *normal* case for typical code, but these attribute rules can vary in classes that utilize more advanced tools you’ll meet later. For example, reference inheritance can be richer than implied here when metaclasses are deployed, and classes that leverage attribute management tools such as properties, descriptors, and `__setattr__` can intercept and route attribute assignments arbitrarily.

In fact, some inheritance *is* run on assignment, too, to locate descriptors with a `__set__` method; such tools override the normal rules for both reference and assignment. We’ll explore attribute management tools in depth in [Chapter 38](#) and formalize inheritance and its use of descriptors in [Chapter 40](#). For now, most readers should focus on the normal rules given here, which cover most Python application code you’re likely to read, write, or run.

The “Zen” of Namespaces: Assignments Classify Names

With distinct search procedures for qualified and unqualified names, and multiple lookup layers for both, it can sometimes be difficult to tell where a name will wind up going. In Python, the place where you *assign* a name is crucial—it fully determines the scope or object in which a name will reside. The file in [Example 29-2](#), *manynames.py*, illustrates how this principle translates to code and summarizes the namespace ideas we have seen throughout this book (sans obscure special-case scopes like comprehensions):

Example 29-2. *manynames.py* (first half)

```
X = 11                                # Global (module) X (manynames.X post import)

def f():
    print(X)                          # Access global X per LEGB lookup

def g():
    X = 22                            # Local (function) X (hides module X)
    print(X)

class C:
    X = 33                            # Class attribute C.X (self.X pre self.m())
    def m(self):
        X = 44                        # Local (function) X in method (unused here)
        self.X = 55                  # Instance attribute self.X (hides class X)
```

This file assigns the same name, *X*, five times—illustrative, though not exactly best practice! Because this name is assigned in five different locations, though, all five *X*s in this program are completely different variables. From top to bottom, the assignments to *X* here generate a module attribute (11), a local variable in a function (22), a class attribute (33), a local variable in a method (44), and an instance attribute (55). Although all five are named *X*, the fact that they are all assigned at different places in the source code or to different objects makes all of these unique variables.

You should study this example carefully because it collects ideas we’ve been exploring throughout the last few parts of this book. When it makes sense to you, you will have achieved Python namespace enlightenment. Or you can run the code and see what happens—[Example 29-3](#) lists the remainder of

the source file in [Example 29-2](#), with self-test code that makes an instance and prints all the Xs that it can fetch.

Example 29-3. *manynames.py* (second half)

```
if __name__ == '__main__':
    print(X)           # 11: module (a.k.a. manynames.X outside file)
    f()                # 11: global
    g()                # 22: local
    print(X)           # 11: module name unchanged

    I = C()            # Make instance
    print(I.X)         # 33: class name inherited by instance
    I.m()              # Attach attribute name X to instance now
    print(I.X)         # 55: instance
    print(C.X)         # 33: class (a.k.a. I.X if no X in I)

    #print(C.m.X)       # FAILS: only visible in method
    #print(g.X)         # FAILS: only visible in function
```

The outputs that are printed when the file is run are noted in the comments in the code; trace through them to see which variable named X is being accessed each time. Notice in particular that we can go through the class to fetch its attribute (C.X), but we can never fetch local variables in functions or methods from outside their def statements. Locals are visible only to other code within the def, and, in fact, only live in memory while a call to the function or method is executing.

Some of the names defined by this file are visible *outside the file* to other modules, too, but recall that we must always import before we can access names in another file—name segregation is the main point of modules, after all. [Example 29-4](#) shows how names appear outside the module in [Example 29-3](#), again with expected outputs in comments:

Example 29-4. *manynames-client.py*

```
import manynames

X = 66
print(X)                # 66: the global here
print(mynames.X)        # 11: globals become attributes after imports

mynames.f()             # 11: manynames's X, not the one here!
mynames.g()             # 22: local in other file's function

print(mynames.C.X)      # 33: attribute of class in other module
I = manynames.C()
print(I.X)              # 33: still from class here
I.m()
print(I.X)              # 55: now from instance!
```

Notice here how `mynames.f()` prints the X in `mynames`, not the X assigned in this file—scopes are always determined by the position of assignments in your source code (i.e., lexically) and are never influenced by what imports what or who imports whom. Also, notice that the instance's own X is not created until we call `I.m()`—attributes, like all variables, spring into existence when assigned, and not before. Normally, we create instance attributes by assigning them in class `__init__` constructor methods, but this isn't the only option.

Finally, as covered in [Chapter 17](#), it's also possible for a function to *change* names outside itself with `global` and `nonlocal` statements—these statements provide write access, but also modify assignment's namespace binding rules. [Example 29-5](#) provides a refresher on these points.

Example 29-5. *funcscope.py*

```
X = 11                                # Global in module

def g1():
    print(X)                          # Reference global in module (11)

def g2():
    global X
    X = 22                            # Change global in module

def h1():
    X = 33                            # Local in function
    def nested():
        print(X)                     # Reference local in enclosing scope (33)

def h2():
    X = 33                            # Local in function
    def nested():
        nonlocal X
        X = 44                      # Change local in enclosing scope
```

Of course, you generally shouldn't use the same name for every variable in your script—but as this example demonstrates, even if you do, Python's namespaces will work to keep names used in one context from accidentally clashing with those used in another.

Nested Classes: The LEGB Scopes Rule Revisited

The preceding example summarized the effect of nested functions on scopes, which we studied in [Chapter 17](#). As we saw briefly near the start of this section, classes can be nested, too—a useful coding pattern in some types of programs. This has scope implications that follow naturally from what you already know, but that may not be obvious on first encounter. This section illustrates the concept by example.

Though they are normally coded at the top level of a module, classes also appear nested in functions that generate them—a variation on the “factory function” (a.k.a. *closure*) theme in [Chapter 17](#), with similar state retention roles. There, we noted that `class` statements introduce new local scopes, much like function `def` statements, which follow the same LEGB scope lookup rule as function definitions.

This rule applies both to the top level of the class itself as well as to the top level of method functions nested within it. Both form the *L* layer in this rule—they are local scopes with access to their names, names in any enclosing functions, globals in the enclosing module, and built-ins. Like modules, the class's local scope *morphs* into an attribute namespace after the `class` statement is run, but its top-level code is a local scope while the `class` runs.

Importantly, though, although classes have access to enclosing functions' scopes, they do not themselves act as enclosing scopes to code nested within the class—Python searches enclosing functions for referenced names but *never* any enclosing classes. That is, a class *is* a local scope and has access *to* enclosing local scopes, but it does not *serve* as an enclosing local scope to further nested code. Because

the search for names used in method functions skips the enclosing class, class attributes must be fetched as object attributes using inheritance.

For example, in the `nester` function of [Example 29-6](#), all references to `X` are routed to the global scope except the last, which picks up a local-scope redefinition in `method2` (the output of each example in this section is described in its last two comments).

Example 29-6. classscope1.py

```
X = 1

def nester():
    print(X)                # Global: 1
    class C:
        print(X)            # Global: 1
        def method1(self):
            print(X)         # Global: 1
        def method2(self):
            X = 3             # Hides global
            print(X)         # Local: 3
    I = C()
    I.method1()
    I.method2()

print(X)                    # Global: 1
nester()                    # Rest: 1, 1, 1, 3
```

Watch what happens, though, when we reassign the same name in nested function layers in [Example 29-7](#): the redefinitions of `X` create locals that hide those in enclosing scopes, just as for simple nested functions; the enclosing class layer does not change this rule, and in fact is irrelevant to it.

Example 29-7. classscope2.py

```
X = 1

def nester():
    X = 2                    # Hides global
    print(X)                 # Local: 2
    class C:
        print(X)             # In enclosing def (nester): 2
        def method1(self):
            print(X)          # In enclosing def (nester): 2
        def method2(self):
            X = 3              # Hides enclosing (nester)
            print(X)          # Local: 3
    I = C()
    I.method1()
    I.method2()

print(X)                    # Global: 1
nester()                    # Rest: 2, 2, 2, 3
```

Finally, [Example 29-8](#) shows what happens when we reassign the same name at multiple stops along the way: assignments in the local scopes of both functions and classes hide globals or enclosing function locals of the same name, regardless of the nesting involved.

Example 29-8. classscope3.py

```
X = 1

def nester():
    X = 2                # Hides global
    print(X)             # Local: 2
    class C:
        X = 3            # Class local hides nester's: C.X or I.X (not scoped)
        print(X)         # Local: 3
        def method1(self):
            print(X)      # In enclosing def (not 3 in class!): 2
            print(self.X) # Inherited class local: 3
        def method2(self):
            X = 4         # Hides enclosing (nester, not class)
            print(X)      # Local: 4
            self.X = 5    # Hides class's
            print(self.X) # Located in instance: 5
    I = C()
    I.method1()
    I.method2()

print(X)                # Global: 1
nester()                # Rest: 2, 3, 2, 3, 4, 5
```

Most importantly, the lookup rules for simple names like `X` never search enclosing class statements—just defs, modules, and built-ins (it's the LEGB rule, not LCEGB!). In `method1`, for example, `X` is found in a `def` outside the enclosing class that has the same name in its local scope. To get to names assigned in the class (e.g., methods), we must fetch them as class or instance object attributes, via `self.X` in this case.

Believe it or not, you'll see valid roles for this nested-classes coding pattern later in this book, especially in some of [Chapter 39's](#) *decorators*. In this role, the enclosing function usually both serves as a class or instance factory and provides retained state for later use in the enclosed class or its methods.

Namespace Dictionaries: Review

In [Chapter 23](#), we saw that module namespaces have a concrete implementation as dictionaries, exposed with the built-in `__dict__` attribute. In [Chapters 27](#) and [28](#), we saw that the same holds true for class and instance objects—attribute qualification is mostly a dictionary indexing operation internally, and attribute inheritance is largely a matter of searching linked dictionaries. In fact, within Python, instance and class objects are mostly just dictionaries with links between them. Python exposes these dictionaries, as well as their links, for use in advanced roles.

We put some of these tools to work in the prior chapter, but to summarize and help you better understand how attributes work internally, let's work through an interactive session that traces the way namespace dictionaries grow when classes are involved. Now that we know more about methods and superclasses, we can also embellish the coverage here for a better look. First, let's define a superclass and a subclass with methods that will store data in their instances:

```
>>> class Super:
    def hello(self):
        self.data1 = 'hack'

>>> class Sub(Super):
    def hola(self):
        self.data2 = 'code'
```

When we make an instance of the subclass, the instance starts out with an empty namespace dictionary, but it has links back to the class for the inheritance search to follow. In fact, the inheritance tree is explicitly available in special attributes, which you can inspect. Instances have a `__class__` attribute that links to their class, and classes have a `__bases__` attribute that is a tuple containing links to higher superclasses:

```
>>> X = Sub()
>>> X.__dict__                                # Instance namespace dict
{}
>>> X.__class__                               # Class of instance
<class '__main__.Sub'>
>>> Sub.__bases__                             # Superclasses of class
(<class '__main__.Super'>,)
>>> Super.__bases__                           # Implied above top-levels
(<class 'object'>,)

```

As classes assign to `self` attributes, they populate the instance objects—that is, attributes wind up in the instances' attribute namespace dictionaries, not in the classes'. An instance object's namespace records data that can vary from instance to instance, and `self` is a hook into that namespace:

```
>>> Y = Sub()

>>> X.hello()
>>> X.__dict__
{'data1': 'hack'}

>>> X.hola()
>>> X.__dict__
{'data1': 'hack', 'data2': 'code'}

>>> list(Sub.__dict__.keys())
['__module__', 'hola', '__doc__']
>>> list(Super.__dict__.keys())
['__module__', 'hello', '__dict__', '__weakref__', '__doc__']

>>> Y.__dict__
{}

```

Notice the extra underscore names in the class dictionaries; Python sets these automatically, and we can filter them out with the generator expressions we coded in Chapters 27 and 28 omitted here for space. Most are not used in typical programs but may be used by tools (e.g., `__doc__` holds the docstrings discussed in Chapter 15).

Also, observe that `Y`, a second instance made at the start of this series, still has an empty namespace dictionary at the end, even though `X`'s dictionary has been populated by assignments in methods. Again, each instance has an independent namespace dictionary, which starts out empty and can record completely different attributes than those recorded by the namespace dictionaries of other instances of the same class.

Because instance attributes are actually dictionary keys inside Python, there are really two ways to fetch and assign their values—by qualification or by key indexing:

```
>>> X.data1, X.__dict__['data1']
('hack', 'hack')

>>> X.data3 = 'docs'
>>> X.__dict__
{'data1': 'hack', 'data2': 'code', 'data3': 'docs'}

```

```
>>> X.__dict__['data3'] = 'apps'
>>> X.data3
'apps'
```

This equivalence applies only to attributes actually attached to the *instance*, though. Because attribute fetch qualification also performs an inheritance search, it can access *inherited* attributes that namespace dictionary indexing cannot. The inherited attribute `X.hello`, for instance, cannot be accessed by `X.__dict__['hello']`.

Experiment with these special attributes on your own to get a better feel for how namespaces actually do their attribute business. Also, try running these objects through the `dir` function we met in the prior two chapters—`dir(X)` is similar to `X.__dict__.keys()`, but `dir` sorts its list and includes inherited attributes. Even if you will never use these in the kinds of programs you write, seeing how attributes are stored can help solidify namespaces in general.



The slots exception: In [Chapter 32](#), you'll learn about *slots*, an advanced class tool that stores attributes in instances but not in their namespace dictionaries. It's tempting to treat these as class attributes, and indeed, they appear in *class* namespaces where they manage per-instance values. As you'll find, though, slots may prevent a `__dict__` from being created in the instance—a potential that generic tools must sometimes account for by using storage-neutral built-ins like `dir` to list and `getattr` to fetch. The good news is that slots are used very rarely—as they should be!

Namespace Links: A Tree Climber

The prior section demonstrated the special `__class__` and `__bases__` instance and class attributes without really explaining why you might care about them. In short, these attributes allow you to inspect inheritance hierarchies within your own code. For example, they can be used to display a class tree, as coded in the module of [Example 29-9](#).

Example 29-9. classtree.py

```
"""
classtree.py: Climb inheritance trees using namespace links,
displaying higher superclasses with indentation for height
"""

def classtree(cls, indent):
    print('.' * indent + cls.__name__)    # Print class name here
    for supercls in cls.__bases__:        # Recur to all superclasses
        classtree(supercls, indent+3)    # May visit super > once

def instancetree(inst):
    print('Tree of', inst)                # Show instance
    classtree(inst.__class__, 3)          # Climb to its class

def selftest():
    class A:    pass
    class B(A): pass
    class C(A): pass
    class D(B,C): pass
    class E:    pass
    class F(D,E): pass
```

```

    instancetree(B())
    instancetree(F())

if __name__ == '__main__': selftest()

```

The `classtree` function in this script is *recursive*—it prints a class’s name using `__name__`, then climbs up to the superclasses by calling itself. This allows the function to traverse arbitrarily shaped class trees; the recursion climbs to the top and stops at root superclasses that have empty `__bases__` attributes. As explored in [Chapter 19](#), when using recursion, each active level of a function gets its own copy of the local scope. Here, this means that `cls` and `indent` are different at each `classtree` level.

Most of this file is self-test code. When run standalone, it builds an empty class tree, makes two instances from it, and prints their class tree structures. The trees include the implied object superclass that is automatically added above standalone root (i.e., topmost) classes; there’s more on object in [Chapter 32](#):

```

$ python3 classtree.py
Tree of <__main__.selftest.<locals>.B object at 0x10733a000>
...B
.....A
.....object
Tree of <__main__.selftest.<locals>.F object at 0x10733a000>
...F
.....D
.....B
.....A
.....object
.....C
.....A
.....object
.....E
.....object

```

Here, indentation marked by periods is used to denote class tree height. Of course, we could improve on this output format and perhaps even sketch it in a GUI display. Even as is, though, we can import these functions anywhere we want a quick display of a physical class tree:

```

$ python3
>>> class Employee: pass
>>> class Person(Employee): pass
>>> pat = Person()

>>> import classtree
>>> classtree.instancetree(pat)
Tree of <__main__.Person object at 0x1072a1b80>
...Person
.....Employee
.....object

```

Regardless of whether you will ever code or use such tools, this example demonstrates one of the many ways that you can make use of special attributes that expose interpreter internals. You’ll see others when we code general-purpose class display tools in [“Multiple Inheritance and the MRO” on page 796](#)—there, we will extend this technique to also display attributes in each object in a class tree and function as a reusable superclass.

In the last part of this book, we’ll revisit such tools in the context of Python tool building at large, to code tools that implement attribute privacy, argument validation, and more. While not in every Python programmer’s job description, access to internals enables powerful development tools.

Documentation Strings Revisited

The last section's example includes a docstring for its module, but remember that docstrings can be used for class components as well. Docstrings, which we covered in detail in [Chapter 15](#), are string literals that show up at the top of various structures and are automatically saved by Python in the corresponding objects' `__doc__` attributes. This works for module files, function defs, and classes and methods.

Now that we've seen more about classes and methods, [Example 29-10](#), a.k.a. *docstr.py*, provides a quick but comprehensive example that summarizes the places where docstrings can show up in your code. All of these can be triple-quoted blocks or simpler one-liner literals like those here.

Example 29-10. *docstr.py*

```
"I am: docstr.__doc__"

def func(args):
    "I am: docstr.func.__doc__"
    pass

class Klass:
    "I am: Klass.__doc__ or docstr.Klass.__doc__ or self.__doc__"
    def method(self):
        "I am: Klass.method.__doc__ or self.method.__doc__"
        print(self.__doc__)
        print(self.method.__doc__)
```

The main advantage of documentation strings is that they stick around at runtime. Thus, if it's been coded as a docstring, you can qualify an object with its `__doc__` attribute to fetch its documentation (calling `print` on the result interprets line breaks if it's a multiline string):

```
$ python3
>>> import docstr

>>> docstr.__doc__
'I am: docstr.__doc__'
>>> docstr.func.__doc__
'I am: docstr.func.__doc__'
>>> docstr.Klass.__doc__
'I am: Klass.__doc__ or docstr.Klass.__doc__ or self.__doc__'
>>> docstr.Klass.method.__doc__
'I am: Klass.method.__doc__ or self.method.__doc__'

>>> x = docstr.Klass()
>>> x.method()
I am: Klass.__doc__ or docstr.Klass.__doc__ or self.__doc__
I am: Klass.method.__doc__ or self.method.__doc__
```

A discussion of the *PyDoc* tool, which knows how to format all these strings in reports and web pages, appears in [Chapter 15](#). Here it is running its `help` function on our code:

```
>>> help(docstr)
Help on module docstr:

NAME
    docstr - I am: docstr.__doc__

CLASSES
```



```

builtins.object
    Klass

class Klass(builtins.object)
| I am: Klass.__doc__ or docstr.Klass.__doc__ or self.__doc__
|
| Methods defined here:
|
| method(self)
|     I am: Klass.method.__doc__ or self.method.__doc__
|
| -----
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables
|
| __weakref__
|     list of weak references to the object
|

FUNCTIONS
    func(args)
        I am: docstr.func.__doc__

FILE
    /.../LP6E/Chapter29/docstr.py

```

Documentation strings are available at runtime, but they are less flexible syntactically than `#` comments, which can appear anywhere in a program. Both forms are useful, and any program documentation is good (as long as it's accurate, of course!). As stated before, the Python “best practice” rule of thumb is to use docstrings for higher-level functional documentation and hash-mark comments for more fine-grained coding documentation.

Classes Versus Modules

Finally, let's wrap up this chapter by briefly comparing the topics of this book's last two parts: modules and classes. Because they're both about namespaces, the distinction can be confusing. In short:

Modules

- Implement data+logic packages
- Are created with Python files or other-language extensions
- Are used by being imported
- Form the top level in Python program structure

Classes

- Implement new full-featured objects
- Are created with `class` statements
- Are used by being called
- Always live within a module

Classes also support extra features that modules don't, such as operator overloading, multiple instance generation, and inheritance. Although both classes and modules are namespaces, you should be able to tell by now that they are very different things. We need to move ahead to see just how unique classes can be.

Chapter Summary

This chapter took us on a second, more in-depth tour of the OOP mechanisms of the Python language. We learned more about classes, methods, and inheritance, and we wrapped up the namespaces and scopes story in Python by extending it to cover its application to classes. Along the way, we encountered core OOP concepts such as abstract superclasses, class data attributes, namespace links, and manual calls to superclass methods and constructors.

Now that we've explored all the basic mechanics of coding classes in Python, the next chapter turns to a specific facet of those mechanics: *operator overloading*. After that, we'll explore common design patterns, looking at some of the ways that classes are commonly used and combined to optimize code reuse. Before you read ahead, though, be sure to work through the usual chapter quiz to review what we've covered here.

Test Your Knowledge: Quiz

1. What is an abstract superclass?
2. What happens when a simple assignment statement appears at the top level of a `class` statement?
3. Why might a class need to manually call the `__init__` method in a superclass?
4. How can you augment, instead of completely replacing, an inherited method?
5. How does a class's local scope differ from that of a function?

Test Your Knowledge: Answers

1. An abstract superclass is a class that calls a method, but does not inherit or define it—it expects the method to be filled in by a subclass. This is often used as a way to generalize classes when behavior cannot be predicted until a more specific subclass is coded. OOP frameworks also use this as a way to dispatch to client-defined, customizable operations.
2. When a simple assignment statement (`X = Y`) appears at the top level of a `class` statement, it attaches a data attribute to the class (`Class.X`). Like all class attributes, this will be shared by all instances that do not have the same attribute. Methods are generally created instead by `def` statements nested in a `class`.
3. A class must manually call the `__init__` method in a superclass if it defines an `__init__` constructor of its own and still wants the superclass's construction code to run (and it often will). Python itself automatically runs just *one* constructor—the lowest one in the inheritance tree. Superclass constructors are often called through the class name, passing in the `self` instance manually: `Superclass.__init__(self, ...)`; they may also be called by `super().__init__(...)`, though we haven't yet studied this form in full.

4. To augment instead of completely replacing an inherited method, redefine it in a subclass, but call back to the superclass's version of the method manually from the new version of the method in the subclass. That is, pass the `self` instance to the superclass's version of the method manually: `Superclass.method(self, ...)`; or do so implicitly with `super().method(...)`. The prior answer is really just a special case of this one.
5. A class is a local scope and has access to enclosing local scopes, but it does not serve as an enclosing local scope to further nested code. Like modules, the class local scope morphs into an attribute namespace after the `class` statement is run.

Operator Overloading

This chapter continues our in-depth survey of class mechanics by focusing on operator overloading. We looked briefly at operator overloading in prior chapters. Here, we'll fill in more details and explore a handful of commonly used overloading methods, most of which we haven't yet encountered. Although we don't have space to demonstrate each of the many operator-overloading methods available, those we will code here are a representative sample large enough to uncover the possibilities of this Python class feature.

The Basics

Really “operator overloading” simply means *intercepting* built-in operations in a class's methods—Python automatically invokes your methods when instances of the class appear in built-in operations, and your method's return value becomes the result of the corresponding operation. Here's a review of the key ideas behind overloading:

- Operator overloading lets classes intercept normal Python operations.
- Classes can overload all Python built-in expression operators.
- Classes can also overload other built-in operations such as printing, function calls, and attribute access.
- Overloading is implemented by providing specially named methods in a class.
- Python predefines the special method names that correspond to built-in operations.

In other words, when methods of predefined special names are provided in a class, Python automatically calls them when instances of the class appear in their associated built-in operations or expressions. Your class provides the behavior of the corresponding operation for instance objects created from it.

As you've learned, operator-overloading methods are never required and generally don't have defaults (apart from a handful that all classes get from the implied object root class). If you don't code or inherit an overloading method, it just means that your class does not support the corresponding operation. When used, though, these methods allow classes to emulate the interfaces of built-in objects, which makes them consistent, and compatible with more code.

Constructors and Expressions: `__init__` and `__sub__`

As a warm-up, consider the simple class in [Example 30-1](#): its `Number` class, coded in module file `number.py`, provides a method to intercept instance construction (`__init__`), as well as one for catching subtraction expressions (`__sub__`). Special methods like these are the hooks that let you tie into built-in operations.

Example 30-1. number.py

```
class Number:
    def __init__(self, start):                # On Number(start)
        self.data = start
    def __sub__(self, other):                 # On instance - other
        return Number(self.data - other)     # Result is a new instance
```

As we've already learned, the `__init__` constructor method seen in this code is the most commonly used operator-overloading method in Python; it's present in most classes and used to initialize the newly created instance object using any arguments passed to the class name. The `__sub__` method plays the binary-operator role that `__add__` did in [Chapter 27](#)'s introduction, intercepting subtraction expressions and returning a new instance of the class as its result (and running `__init__` along the way):

```
>>> from number import Number              # Fetch class from module
>>> X = Number(5)                           # Number.__init__(X, 5)
>>> Y = X - 2                               # Number.__sub__(X, 2)
>>> Y.data                                  # Y is new Number instance
3
```

We've already studied `__init__` and basic binary operators like `__sub__` in some depth, so we won't rehash their usage further here. In this chapter, we will tour some other tools available in this domain and look at example code that applies them in common use cases.



Construction convolution: Technically, instance creation first triggers the `__new__` method, which creates and returns the new instance object, which is then passed into `__init__` for initialization. Since `__new__` has a built-in implementation and is redefined in only very limited roles, though, nearly all Python classes initialize by defining an `__init__` method. We'll explore one use case for `__new__` when we study *metaclasses* in [Chapter 40](#); though rare, it is sometimes also used to customize creation of instances of immutable types.

Common Operator-Overloading Methods

Just about everything you can do to built-in objects such as integers and lists has a corresponding specially named method for overloading in classes. [Table 30-1](#) lists a few of the most common; there are many more. In fact, many overloading methods come in multiple versions (e.g., `__add__`, `__radd__`, and `__iadd__` for addition), which is one reason there are so many. See the Python language reference manual for an exhaustive list of the special method names available.

Table 30-1. Common operator-overloading methods

Method	Implements	Called for
<code>__init__</code>	Constructor	Object creation: <code>X = Class(args)</code>
<code>__del__</code>	Destructor	Object reclamation of <code>X</code>
<code>__add__</code>	Operator <code>+</code> (among others)	<code>X + Y</code> , <code>X += Y</code> if no <code>__iadd__</code>
<code>__or__</code>	Operator <code> </code> (bitwise OR)	<code>X Y</code> , <code>X = Y</code> if no <code>__ior__</code>
<code>__repr__</code> , <code>__str__</code>	Printing, conversions	<code>print(X)</code> , <code>X</code> , <code>repr(X)</code> , <code>str(X)</code> , <code>f' {X!r} '</code>
<code>__call__</code>	Function calls	<code>X(*pargs, **kargs)</code>
<code>__getattr__</code>	Attribute fetch	<code>X.undefined</code>
<code>__setattr__</code>	Attribute assignment	<code>X.any = value</code>
<code>__delattr__</code>	Attribute deletion	<code>del X.any</code>
<code>__getattribute__</code>	Attribute fetch	<code>X.any</code>
<code>__getitem__</code>	Indexing, slicing, iteration	<code>X[i]</code> , <code>X[i:j]</code> , for and other iterations if no <code>__iter__</code>
<code>__setitem__</code>	Index and slice assignment	<code>X[i] = value</code> , <code>X[i:j] = iterable</code>
<code>__delitem__</code>	Index and slice deletion	<code>del X[i]</code> , <code>del X[i:j]</code>
<code>__len__</code>	Length	<code>len(X)</code> , truth tests if no <code>__bool__</code>
<code>__bool__</code>	Boolean tests	<code>bool(X)</code> , truth tests
<code>__lt__</code> , <code>__gt__</code> , <code>__le__</code> , <code>__ge__</code> , <code>__eq__</code> , <code>__ne__</code>	Comparisons	<code>X < Y</code> , <code>X > Y</code> , <code>X <= Y</code> , <code>X >= Y</code> , <code>X == Y</code> , <code>X != Y</code>
<code>__radd__</code>	Right-side operators	<code>other + X</code>
<code>__iadd__</code>	In-place augmented operators	<code>X += Y</code> (or else <code>__add__</code>)
<code>__iter__</code> , <code>__next__</code>	Iteration tools	<code>I=iter(X)</code> , <code>next(I)</code> , for and other iterations, in if no <code>__contains__</code>
<code>__contains__</code>	Membership test	<code>item in X</code>
<code>__index__</code>	Integer value	<code>hex(X)</code> , <code>bin(X)</code> , <code>oct(X)</code> , <code>0[X]</code> , <code>0[X:]</code>
<code>__enter__</code> , <code>__exit__</code>	Context manager (Chapter 34)	<code>with obj as var:</code>
<code>__get__</code> , <code>__set__</code> , <code>__delete__</code>	Descriptor attributes (Chapter 38)	<code>X.attr</code> , <code>X.attr = value</code> , <code>del X.attr</code>
<code>__new__</code>	Creation (Chapter 40)	Object creation, before <code>__init__</code>

All overloading methods have names that start and end with two *underscores* to keep them distinct from other names you define in your classes. The mappings from special method names to expressions or operations are *predefined* by the Python language and documented in full in its standard language manual. For example, the name `__add__` always maps to `+` expressions by Python language definition, regardless of what an `__add__` method's code actually does; it's largely just a dispatch mechanism.

Operator-overloading methods may be *inherited* from superclasses if not defined, just like any other methods. Operator-overloading methods are also all *optional*—if you don't code or inherit one, that operation is simply unsupported by your class, and attempting it will raise an exception. Some built-in operations, like printing, have defaults inherited from the implied object root class, but most built-ins fail for class instances if no corresponding operator-overloading method is present.

As we go along here, keep in mind that most overloading methods are used only in advanced programs that require objects to behave like built-ins, though the `__init__` constructor we’ve already met tends to appear in most classes. With that qualifier, let’s explore some of the additional methods in [Table 30-1](#) by example.



Measure twice, post once: Although expressions trigger operator methods, be careful not to assume that there is a speed advantage to cutting out the middle-person and calling the operator method directly. In fact, calling the operator method directly might *be twice as slow*, presumably because of the overhead of a function call, which Python avoids or optimizes in built-in cases.

Here’s the story for `len` and `__len__` using [Chapter 21](#)’s timing techniques on Python 3.12 and macOS. Calling `__len__` directly takes twice as long (and has since Python 2.X):

```
$ python3 -m timeit -n 10000 -r 10 \
    -s "L = list(range(100))" "x = L.__len__()"
10000 loops, best of 10: 53.4 nsec per loop

$ python3 -m timeit -n 10000 -r 10 \
    -s "L = list(range(100))" "x = len(L)"
10000 loops, best of 10: 25.3 nsec per loop
```

This is not as contrived as it may seem—recommendations for using the slower alternative in the name of speed have been known to crop up in venues that shall remain nameless here.

Indexing and Slicing: `__getitem__` and `__setitem__`

Our first new method set allows your classes to mimic some of the behaviors of sequences and mappings. If defined in a class (or inherited by it), the `__getitem__` method is called automatically for instance-indexing operations. When an instance `X` appears in an indexing expression like `X[i]`, Python calls the `__getitem__` method inherited by the instance, passing `X` to the first argument and the index `i` in brackets to the second argument.

For example, the following class returns the square of an index value—atypical perhaps but illustrative of the mechanism in general:

```
>>> class Indexer:
    def __getitem__(self, index):
        return index ** 2

>>> X = Indexer()
>>> X[2]                                     # X[i] calls X.__getitem__(i)
4

>>> for i in range(5):
    print(X[i], end=' ')                    # Runs __getitem__(X, i) each time

0 1 4 9 16
```

It’s up to your class to define what this expression means, though it should generally imitate a sequence index or mapping key fetch; returning the index squared as done here works but probably won’t qualify as “best practice.”

Intercepting Slices

Surprisingly, in addition to indexing, `__getitem__` is also called for *slice expressions*. Formally speaking, built-in object types handle slicing the same way. For example, the following demos slicing at work on a built-in list, using upper and lower bounds, omitted parts, and a stride (see [Chapter 7](#) if you need a refresher on slicing):

```
>>> L = [5, 6, 7, 8, 9]
>>> L[2:4]                                # Slice with slice syntax: 2..(4-1)
[7, 8]
>>> L[1:]
[6, 7, 8, 9]
>>> L[:-1]
[5, 6, 7, 8]
>>> L[:2]
[5, 7, 9]
```

Really, though, slicing bounds are bundled up into a *slice object* and passed to the list's implementation of indexing. In fact, you can always pass a slice object manually—slice syntax is mostly syntactic sugar for indexing with a slice object:

```
>>> L[slice(2, 4)]                        # Slice with slice objects
[7, 8]
>>> L[slice(1, None)]
[6, 7, 8, 9]
>>> L[slice(None, -1)]
[5, 6, 7, 8]
>>> L[slice(None, None, 2)]
[5, 7, 9]
```

This matters in classes with a `__getitem__` method—this method will be called *both* for basic indexing (with an index or key) and for slicing (with a slice object). Our previous class won't handle slicing because its math assumes integer indexes are passed, but the following class will. When called for indexing, the argument is an integer as before:

```
>>> class Indexer:
    def __init__(self, data):
        self.data = data
    def __getitem__(self, index):          # Called for index or slice
        print('getitem:', index)
        return self.data[index]          # Perform index or slice

>>> X = Indexer([5, 6, 7, 8, 9])
>>> X[0]                                  # Indexing sends __getitem__ an integer
getitem: 0
5
>>> X[1]
getitem: 1
6
>>> X[-1]
getitem: -1
9
```

When called for *slicing*, though, the method receives a slice object, which is simply passed along to the embedded list indexer in a new index expression:

```

>>> X[2:4]                                     # Slicing sends __getitem__ a slice object
getitem: slice(2, 4, None)
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, -1, None)
[5, 6, 7, 8]
>>> X[::2]
getitem: slice(None, None, 2)
[5, 7, 9]

```

Where needed, `__getitem__` can test the type of its argument, and extract slice object bounds—slice objects have attributes `start`, `stop`, and `step`, any of which can be `None` if omitted:

```

>>> class Indexer:
    def __getitem__(self, index):
        if isinstance(index, int):
            print('indexing', index)
        else:
            print('slicing', index.start, index.stop, index.step)

>>> X = Indexer()
>>> X[99]
indexing 99
>>> X[1:99:2]
slicing 1 99 2
>>> X[1:]
slicing 1 None None

```

Run a `help(slice)` in a REPL for more info on this very special-case built-in object type, and see the section “Membership: `__contains__`, `__iter__`, and `__getitem__`” for another example of slice interception at work.

Intercepting Item Assignments

If used, the `__setitem__` index assignment method similarly intercepts both index and slice assignments—it receives a slice object for the latter, which may be passed along in another index assignment or used directly in the same way:

```

>>> class IndexSetter:
    def __init__(self, data):
        self.data = data
    def __setitem__(self, index, value):
        print('setitem:', index)
        self.data[index] = value

>>> X = IndexSetter([5, 6, 7, 8, 9])
>>> X[0] = 555
setitem: 0
>>> X[-2:] = [888, 999, 111]
setitem: slice(-2, None, None)

>>> X.data
[555, 6, 7, 888, 999, 111]

```

In fact, `__getitem__` may be called automatically in even more contexts than indexing and slicing—it’s also an *iteration* fallback option, as you’ll see in a moment. First, though, let’s clear up a potential point of confusion in this category.

But `__index__` Means As-Integer

Don’t mistake the (perhaps unfortunately named) `__index__` method for `__getitem__` index interception. The `__index__` method returns an *integer value* for an instance when one is needed—and in retrospect, might have been better named `__asindex__`. For example, it’s used by built-ins that convert to digit strings:

```
>>> class C:
    def __index__(self):
        return 255

>>> X = C()
>>> hex(X)           # Integer value
'0xff'
>>> bin(X)
'0b11111111'
>>> oct(X)
'0o377'
```

Although this method does not intercept instance indexing like `__getitem__`, it is also used in contexts that require an integer—including indexing and slicing:

```
>>> eds = [f'LP{i}e' for i in range(256)]
>>> eds[255]
'LP255e'
>>> X = C()
>>> eds[X]           # As index (not X[i]!)
'LP255e'
>>> eds[X:]          # As index (not X[i:]!)
['LP255e']
```

Though arguably misnamed, there is a rich history of former methods that `__index__` subsumes, which we’ll mercifully omit here. The `__getitem__` method also subsumes former tools but retains a fallback role up next.

Index Iteration: `__getitem__`

Our next hook isn’t always obvious to beginners but turns out to be surprisingly useful. In the absence of the more specific iteration methods we’ll get to in the next section, the `for` statement works by repeatedly indexing an object from zero to higher indexes, until an out-of-bounds `IndexError` exception is detected. Because of that, `__getitem__` also turns out to be one way to overload *iteration* in Python—if only this method is defined, `for` loops call the class’s `__getitem__` each time through, with successively higher offsets.

It’s a case of “code one, get one free”—any built-in or user-defined object that responds to indexing also responds to `for` loop iteration:

```
>>> class StepperIndex:
    def __getitem__(self, i):
        return self.data[i]

>>> X = StepperIndex()           # X is a StepperIndex object
```

```

>>> X.data = 'hack'
>>>
>>> X[1]                                # Indexing calls __getitem__
'a'
>>> for item in X:                       # for loops call __getitem__
    print(item, end=' ')               # for indexes items 0..N

h a c k

```

In fact, it's really a case of “code one, get a bunch free.” Any class that supports for loops automatically supports all *iteration tools* in Python, many of which we've explored in earlier chapters (e.g., iteration tools were presented in [Chapter 14](#)). For instance, the `in` membership test, list comprehensions, the `map` built-in, list and tuple assignments, and some type constructors will also call `__getitem__` automatically to iterate if it's defined:

```

>>> 'k' in X                            # All call __getitem__ too
True

>>> [c for c in X]                      # Comprehension
['h', 'a', 'c', 'k']

>>> list(map(str.upper, X))              # map calls
['H', 'A', 'C', 'K']

>>> (a, b, c, d) = X                    # Sequence assignments
>>> a, d
('h', 'k')

>>> list(X), tuple(X), ''.join(X)       # And so on...
(['h', 'a', 'c', 'k'], ('h', 'a', 'c', 'k'), 'hack')

>>> X
<__main__.StepperIndex object at 0x10c4bcc20>

```

In practice, this technique can be used to create objects that provide a sequence interface and to add logic to built-in sequence type operations; we'll revisit this idea when extending built-in types in [Chapter 32](#).

Iterable Objects: `__iter__` and `__next__`

Although the `__getitem__` technique of the prior section works, it's really just a legacy fallback for iteration. Today, all iteration tools in Python will try the `__iter__` method first before trying `__getitem__`. That is, they prefer the *iteration protocol* we learned about in [Chapter 14](#) over repeatedly indexing an object; only if the object does not support the iteration protocol is indexing attempted instead. Generally speaking, you should prefer `__iter__` too—it supports general iteration tools better than `__getitem__` can.

For a review of this model's essentials, see [Figure 14-1](#) in [Chapter 14](#). In brief, iteration tools work by running an *iterable* object's `__iter__` method to fetch an *iterator* object. If this works as planned, Python then repeatedly calls this iterator object's `__next__` method to produce items until it raises a `StopIteration` exception. Built-in functions `iter` and `next` are also available as a conveniences for manual iterations—`iter(X)` is the same as `X.__iter__()` and `next(I)` is the same as `I.__next__()`, and Python internals may vary.

This iterable-object interface is given priority and attempted first. Only if no such `__iter__` method is found, Python falls back on the `__getitem__` scheme and repeatedly indexes by offsets as before until an `IndexError` exception is raised.

User-Defined Iterables

In the `__iter__` scheme, classes implement user-defined iterables by simply implementing the iteration protocol introduced in [Chapter 14](#) and elaborated in [Chapter 20](#). For example, the code in [Example 30-2](#) uses a class to define a user-defined iterable that generates squares on demand, instead of all at once.

Example 30-2. *squares.py*

```
class Squares:
    def __init__(self, start, stop):      # Save state when created
        self.value = start - 1
        self.stop = stop

    def __iter__(self):                  # Return iterator object
        return self                     # Also called by iter() built-in

    def __next__(self):                  # Return a square on each iteration
        if self.value == self.stop:     # Also called by next() built-in
            raise StopIteration
        self.value += 1
        return self.value ** 2
```

When imported, its instances can appear in iteration tools just like built-ins:

```
$ python3
>>> from squares import Squares
>>> for i in Squares(1, 5):           # for calls __iter__
    print(i, end=' ')               # Each iteration calls __next__

1 4 9 16 25
```

Here, the iterator object returned by `__iter__` is simply the instance `self` because the `__next__` method is part of this class itself. In more complex scenarios, the iterator object may be defined as a separate class and object with its own state information to support multiple active iterations over the same instance data (we'll code an example of this in a moment).

The end of the iteration is signaled with a Python `raise` statement—introduced in [Chapter 29](#) and covered in full in the next part of this book, but which simply raises an exception as if Python itself had done so. Because of all this, manual iterations work the same on user-defined iterables as they do on built-in objects:

```
>>> X = Squares(1, 5)                # Iterate manually: what loops do
>>> I = iter(X)                      # iter calls __iter__
>>> next(I)                          # next calls __next__
1
>>> next(I)
4
...more omitted...
>>> next(I)
25
>>> next(I)                          # Can catch this in try statement
StopIteration
```

An equivalent coding of this iterable with `__getitem__` might be less natural because the `for` would then iterate through all offsets zero and higher; the offsets passed in would be only indirectly related to the range of values produced (`0...N` would need to map to `start...stop`). Because `__iter__` objects retain explicitly managed state between next calls, they can be more general than `__getitem__`.

On the other hand, iterables based on `__iter__` can sometimes be more complex and less functional than those based on `__getitem__`. They are really designed for iteration, not random indexing—in fact, they don’t overload the indexing expression at all, though you can collect their items in a sequence such as a list to enable other operations:

```
>>> X = Squares(1, 5)
>>> X[1]
TypeError: 'Squares' object is not subscriptable
>>> list(X)[1]
4
```

Single versus multiple scans

The `__iter__` scheme is also the implementation for all the other iteration tools we saw in action for the `__getitem__` method—membership tests, type constructors, sequence assignment, and so on. Unlike our prior `__getitem__` example, though, we also need to be aware that a class’s `__iter__` may be designed for a *single traversal* only, not many. Classes can choose either behavior explicitly in their code.

For example, because the current `Squares` class’s `__iter__` always returns `self` with just one copy of iteration state, it is a *single-scan* iteration; once you’ve iterated over an instance of that class, it’s empty. Calling `__iter__` again on the same instance returns `self` again—in whatever state it may have been left. As coded, you generally need to make a new iterable instance object for each new iteration:

```
>>> X = Squares(1, 5)                # Make an iterable with state
>>> [n for n in X]                    # Exhausts items: __iter__ returns self
[1, 4, 9, 16, 25]
>>> [n for n in X]                    # Now it's empty: __iter__ returns same self
[]

>>> [n for n in Squares(1, 5)]        # Make a new iterable object
[1, 4, 9, 16, 25]
>>> list(Squares(1, 3))               # A new object for each new __iter__ call
[1, 4, 9]
```

To support multiple iterations more directly, we could also recode this example with an extra class or other technique, as we will in a moment. As is, though, by creating a *new instance* for each iteration, you get a fresh copy of iteration state:

```
>>> 36 in Squares(1, 10)              # Other iteration tools
True
>>> a, b, c = Squares(1, 3)           # Each calls __iter__ and then __next__
>>> a, b, c
(1, 4, 9)
>>> ':'.join(map(str, Squares(1, 5)))
'1:4:9:16:25'
```

Just like single-scan built-ins such as `map`, converting to a *list* supports multiple scans as well but adds time and space performance costs, which may or may not be significant to a given program:

```
>>> X = Squares(1, 5)
>>> tuple(X), tuple(X)                # Iterator exhausted in second tuple()
((1, 4, 9, 16, 25), ())

>>> X = list(Squares(1, 5))
>>> tuple(X), tuple(X)
((1, 4, 9, 16, 25), (1, 4, 9, 16, 25))
```

We'll improve this to support multiple scans more directly ahead after a bit of compare and contrast.

Classes versus generators

Notice that the `Squares` iterable of [Example 30-2](#) that we've been using so far would probably be simpler if it was coded with *generator functions or expressions*—tools introduced in [Chapter 20](#) that automatically produce iterable objects and retain local variable state between iterations:

```
>>> def gsquares(start, stop):          # Generator function
    for i in range(start, stop + 1):
        yield i ** 2

>>> for i in gsquares(1, 5):
    print(i, end=' ')

1 4 9 16 25

>>> for i in (x ** 2 for x in range(1, 6)):    # Generator expression
    print(i, end=' ')

1 4 9 16 25
```

Unlike classes, generator functions and expressions implicitly save their state and create the methods required to conform to the *iteration protocol*—with obvious advantages in code conciseness for simpler examples like these. That is, generators' automatic and implicit `__iter__` and `__next__` suffice here.

On the other hand, the class's more explicit attributes and methods, extra structure, inheritance hierarchies, and support for multiple behaviors may be better suited for richer use cases.

Of course, for this artificial example, you could in fact skip both techniques and simply use a `for` loop, `map`, or a list comprehension to build the list all at once. Barring performance data to the contrary, the best and fastest way to accomplish a task in Python is often also the simplest:

```
>>> [x ** 2 for x in range(1, 6)]
[1, 4, 9, 16, 25]
```

That said, classes are better at modeling more complex iterations, especially when they can benefit from the assets of classes in general. An iterable that produces items in a complex database or web service result, for example, might be able to take fuller advantage of classes—and leverage more flexible coding structures like that of the next section.

Multiple Iterators on One Object

Earlier, it was mentioned in passing that the *iterator* object (with a `__next__`) produced by an iterable may be defined as a separate class with its own state information to more directly support multiple active iterations over the same data. To understand this better, consider what happens when we step across a built-in type like a string:

```
>>> S = 'ace'
>>> for x in S:
    for y in S:
        print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

Here, the outer loop grabs an iterator from the string by calling `iter`, and each nested loop does the same to get an independent iterator. Because each active iterator has its own state information, each loop can maintain its own position in the string, regardless of any other active loops. Moreover, we're not required to make a new string or convert to a list each time; the single string object itself supports multiple scans.

We saw related examples earlier, in Chapters 14 and 20. For instance, generator functions and expressions, as well as built-ins like `map` and `zip`, proved to be single-iterator objects, thus supporting a single active scan. By contrast, the `range` built-in, and other built-in types like lists, support multiple active iterators with independent positions.

When we code user-defined iterables with classes, it's up to us to decide whether we will support a single active iteration or many. To achieve the multiple-iterator effect, `__iter__` simply needs to define a new stateful object for the iterator instead of returning `self` for each iterator request.

For example, the class `SkipObject` in [Example 30-3](#) defines an iterable object that skips every other item on iterations. Because its iterator object is created anew from a supplemental class for each iteration, it supports multiple active loops directly.

Example 30-3. *skipper.py*

```
class SkipObject:
    def __init__(self, wrapped):          # Save item to be used
        self.wrapped = wrapped

    def __iter__(self):
        return SkipIterator(self.wrapped) # New iterator each time

class SkipIterator:
    def __init__(self, wrapped):
        self.wrapped = wrapped          # Iterator state information
        self.offset = 0

    def __next__(self):
        if self.offset >= len(self.wrapped): # Terminate iterations
            raise StopIteration
        else:
            item = self.wrapped[self.offset] # else return and skip
            self.offset += 2
            return item

if __name__ == '__main__':
    alpha = 'abcdef'
    skipper = SkipObject(alpha)          # Make container object
    I = iter(skipper)                    # Make an iterator on it
    print(next(I), next(I), next(I))     # Visit offsets 0, 2, 4

    for x in skipper:                    # for calls __iter__ automatically
        for y in skipper:                # Nested fors call __iter__ again each time
            print(x + y, end=' ')        # Each iterator has its own state, offset
```


When run, this example works like the earlier nested loops with built-in strings. Each active loop has its own position in the string because each obtains an independent iterator object that records its own state information:

```
$ python3 skipper.py
a c e
aa ac ae ca cc ce ea ec ee
```

By contrast, our earlier `Squares` class of [Example 30-2](#) supports just one active iteration unless we call `Squares` again in nested loops to obtain new objects (else the outer `for` loop would run just once). Here, there is just one `SkipObject` iterable, with multiple iterator objects created from it.

Classes versus slices

As before, we could achieve similar results with built-in tools—for example, slicing with a third bound to skip items:

```
>>> S = 'abcdef'
>>> for x in S[::2]:
    for y in S[::2]:          # New objects on each iteration
        print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

This isn't quite the same, though, for two reasons. First, each slice expression here will *physically store* the result list all at once in memory; iterables, on the other hand, produce just one value at a time, which can save substantial space and startup time for large result lists.

Second, slices produce *new objects*, so we're not really iterating over the same object in multiple places here. To be closer to the class, we would need to make a single object to step across by slicing ahead of time:

```
>>> S = 'abcdef'
>>> S = S[::2]
>>> S
'ace'
>>> for x in S:
    for y in S:              # Same object, new iterators
        print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

This is more similar to our class-based solution, but it still stores the slice result in memory all at once (there is no generator form of built-in slicing today), and it's only equivalent for this particular case of skipping every other item.

Because user-defined iterables coded with classes can do anything a class can do, they are much more general than this example may imply. Though such generality is not required in all applications, user-defined iterables are a powerful tool—they allow us to make arbitrary objects look and feel like the other sequences and iterables we have met in this book. We could use this technique with a database object, for example, to support iterations over large database fetches, with multiple cursors into the same query result.

Coding Alternative: `__iter__` Plus `yield`

Now for something more implicit—but potentially useful nonetheless. In some applications, it's possible to minimize coding requirements for user-defined iterables by *combining* the `__iter__` method we're exploring here and the `yield` generator function statement we studied in [Chapter 20](#). Because generator functions *automatically* save local-variable state and create required iterator methods, they fit this role well and complement the state retention and other utility we get from classes.

As a quick review, recall that any function that contains a `yield` statement is turned into a generator function. When called, it returns a new *generator object* with automatic retention of local scope and code position; an automatically created `__iter__` method that simply returns itself; and an automatically created `__next__` method that starts the function or resumes it where it last left off:

```
>>> def gen(x):
    for i in range(x): yield i ** 2

>>> G = gen(5)           # Create a generator with __iter__ and __next__
>>> G.__iter__() is G    # Both methods exist on the same object
True
>>> I = iter(G)          # Runs __iter__: generator returns itself
>>> next(I), next(I)     # Runs __next__
(0, 1)
>>> list(gen(5))         # Iteration tools automatically run iter and next
[0, 1, 4, 9, 16]
```

This is still true even if the generator function with a `yield` happens to be a *method* named `__iter__`. Whenever invoked by an iteration tool, such a method will return a new *generator* object with the requisite `__next__`. As an added bonus, generator functions coded as methods in classes have access to saved state in *both* instance attributes and local-scope variables.

For example, the class in [Example 30-4](#) is equivalent to the initial `Squares` user-defined iterable we coded earlier in [Example 30-2](#), but noticeably shorter (4 lines, for anyone counting).

Example 30-4. `squares_yield.py`

```
class Squares:
    # __iter__ + yield generator
    def __init__(self, start, stop):
        # __next__ is automatic/implied
        self.start = start
        self.stop = stop

    def __iter__(self):
        for value in range(self.start, self.stop + 1):
            yield value ** 2
```

As before, for loops and other iteration tools iterate through instances of this class automatically:

```
$ python3
>>> from squares_yield import Squares
>>> for i in Squares(1, 5): print(i, end=' ')    # Runs __iter__, then __next__

1 4 9 16 25
```

And as always, we can also look under the hood to see how this actually works in iteration tools like `for`. Running our class instance through `iter` obtains the result of calling `__iter__` as usual. In this case, though, the result is a generator object with an automatically created `__next__` of the same sort we always get when calling a generator function that contains a `yield`. The only difference here is that the

generator function is automatically called on `iter`. Invoking the result object's `next` interface produces results on demand:

```
>>> S = Squares(1, 5)           # Runs __init__: class saves instance state
>>> S
<squares_yield.Squares object at 0x109e30b90>

>>> I = iter(S)                 # Runs __iter__: returns a generator
>>> I
<generator object Squares.__iter__ at 0x109ecb3e0>
>>> next(I)
1
>>> next(I)                     # Runs generator's __next__
4
...etc...
>>> next(I)                     # Generator has both instance and local scope state
StopIteration
```

It may also help to notice that we could name the generator method something other than `__iter__` and call manually to iterate—`Squares(...).gen()`, for example. Using the `__iter__` name invoked automatically by iteration tools simply skips a manual attribute fetch and call step. [Example 30-5](#) demos the idea.

Example 30-5. `squares_yield_manual.py`

```
import squares_yield             # Reuse prior example's __init__

class Squares(squares_yield.Squares):    # Non __iter__ equivalent
    def gen(self):
        for value in range(self.start, self.stop + 1):
            yield value ** 2
```

The example also imports [Example 30-4](#) to inherit its constructor. When run, its results are the same, but we must call the `gen` method explicitly to fetch an iterable—a step that `__iter__` automates and obviates:

```
$ python3
>>> from squares_yield_manual import Squares
>>> for i in Squares(1, 5).gen(): print(i, end=' ')
...same results...

>>> S = Squares(1, 5)
>>> I = iter(S.gen())           # Call generator manually for iterable/iterator
>>> next(I)
...same results...
```

Coding the generator as `__iter__` instead cuts out the middleperson in your code, though both schemes ultimately wind up creating a new generator object for each iteration:

- With `__iter__`, iteration triggers `__iter__`, which returns a new generator with `__next__`.
- Without `__iter__`, your code calls to make a generator, which returns itself for `__iter__`.

See [Chapter 20](#) for more on `yield` and generators if this is puzzling, and compare it with the more explicit `__next__` version in [Example 30-2](#) earlier. If you do, you'll notice that the `squares_yield.py` version is 4 lines shorter (7 versus 11, not counting whitespace). In a sense, this scheme reduces class coding requirements much like the closure functions of [Chapter 17](#), but in this case does so with a

combination of functional and OOP techniques instead of an alternative to classes. For example, the generator method still leverages `self` attributes.

This may also seem like one too many levels of *magic* to some observers—it relies on both the iteration protocol and the object creation of generators, both of which are highly implicit (in contradiction of longstanding Python goals). Opinions aside, it's important to understand the non-yield flavor of class iterables too, because it's explicit, general, and sometimes broader in scope.

Still, the `__iter__/yield` technique may prove effective in cases where it applies. It also comes with a substantial advantage—as the next section explains.

Multiple iterators with yield

Besides its code conciseness, the user-defined class iterable of the prior section based upon the `__iter__/yield` combination has an important added bonus—it also supports *multiple active iterators* automatically. This naturally follows from the fact that each call to `__iter__` is a new call to a generator function, which returns a new generator with its own copy of the local scope for state retention. Using [Example 30-4](#) again:

```
$ python3
>>> from squares_yield import Squares    # Using the __iter__/yield Squares
>>> S = Squares(1, 5)
>>> I = iter(S)
>>> next(I); next(I)
1
4
>>> K = iter(S)                          # With yield, multiple iterators automatic
>>> next(K)
1
>>> next(I)                             # I is independent of K: own local state
9
```

Although generator functions are single-scan iterables by nature, the implicit calls to `__iter__` in iteration tools make new generators supporting new independent scans:

```
>>> S = Squares(1, 3)
>>> for i in S:                          # Each "for" calls __iter__
    for j in S:
        print(f'{i}:{j}', end=' ')

1:1 1:4 1:9 4:1 4:4 4:9 9:1 9:4 9:9
```

To do the same without `yield` requires a supplemental class that stores iterator state explicitly and manually, using techniques of the preceding section. Per [Example 30-6](#), this grows to 15 lines: 8 more than with `yield`.

Example 30-6. `squares_nonyield.py`

```
class Squares:
    def __init__(self, start, stop):      # Non-yield generator
        self.start = start               # Multiscans: extra object
        self.stop = stop

    def __iter__(self):
        return SquaresIter(self.start, self.stop)

class SquaresIter:
    def __init__(self, start, stop):
```

```

        self.value = start - 1
        self.stop = stop

    def __next__(self):
        if self.value == self.stop:
            raise StopIteration
        self.value += 1
        return self.value ** 2

```

This works the same as the `yield multiscan` version, but with more—and more explicit—code:

```

$ python3
>>> from squares_nonyield import Squares
>>> for i in Squares(1, 5): print(i, end=' ')

1 4 9 16 25

>>> S = Squares(1, 5)
>>> I = iter(S)
>>> next(I); next(I)
1
4
>>> K = iter(S)                                # Multiple iterators without yield
>>> next(K)
1
>>> next(I)
9

>>> S = Squares(1, 3)
>>> for i in S:                                # Each "for" calls __iter__
    for j in S:
        print(f'{i}:{j}', end=' ')

1:1 1:4 1:9 4:1 4:4 4:9 9:1 9:4 9:9

```

Finally, the generator-based approach could similarly remove the need for an extra iterator class in the prior item-skipper example, *skipper.py* of [Example 30-3](#), thanks to its automatic methods and local variable state retention. [Example 30-7](#) codes the mod, which checks in at 9 lines versus the original's 16, sans the original's self-test.

Example 30-7. *skipper_yield.py*

```

class SkipObject:                                # Another __iter__ + yield generator
    def __init__(self, wrapped):                 # Instance scope retained normally
        self.wrapped = wrapped                  # Local scope state saved auto

    def __iter__(self):
        offset = 0
        while offset < len(self.wrapped):
            item = self.wrapped[offset]
            offset += 2
            yield item

```

This works the same as the `non-yield multiscan` version, but with less—and less explicit—code:

```

$ python3
>>> from skipper_yield import SkipObject
>>> skipper = SkipObject('abcdef')
>>> I = iter(skipper)
>>> next(I); next(I); next(I)

```

```

'a'
'c'
'e'
>>> for x in skipper:                # Each "for" calls __iter__: new auto generator
    for y in skipper:
        print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee

```

Of course, these are all artificial examples that could be replaced with simpler tools like comprehensions, and their code may or may not scale up in kind to more realistic tasks. Study these alternatives to see how they compare. As so often in programming, the best tool for the job will likely be the best tool for your job.

Membership: `__contains__`, `__iter__`, and `__getitem__`

The iteration story is even richer than told thus far. Operator overloading is often *layered*: classes may provide specific methods or more general alternatives used as fallback options. We've already seen this for general iteration (`__iter__` or else `__getitem__`), and will encounter another example ahead when we meet Boolean values.

Also in the iterations domain, classes can implement the `in` membership operator as an iteration, using either the `__iter__` or `__getitem__` methods. To support more specific membership, though, classes may code a `__contains__` method—when present, this method is preferred over `__iter__`, which is preferred over `__getitem__`. The `__contains__` method should define membership as applying to keys for a *mapping* (and can use quick lookups) and as a search for *sequences*.

Consider the class `Iters` in [Example 30-8](#). It codes all three methods and tests membership and various iteration tools applied to an instance. To demo, its methods print trace messages when called, its self-test code cycles through tests coded with `match` to call them out, and its methods' traces show up before those of its tests.

Example 30-8. contains.py

```

def trace(msg, end=''):
    print(f'{msg} ', end=end)                # print sans newline

class Iters:
    def __init__(self, value):
        self.data = value

    def __getitem__(self, i):                 # Fallback for iteration
        trace(f'@get[{i}]')                 # Also for index, slice
        return self.data[i]

    def __iter__(self):                     # Preferred for iteration
        trace('@iter')                       # Allows only one active iterator
        self.ix = 0
        return self

    def __next__(self):
        trace('@next')
        if self.ix == len(self.data): raise StopIteration
        item = self.data[self.ix]
        self.ix += 1
        return item

```

```

def __contains__(self, x):                                # Preferred for 'in' membership
    trace('@contains')
    return x in self.data

def self_test(Iter):
    X = Iter([1, 2, 3, 4])                                # Make one instance
    tests = 'In', 'For', 'Comp', 'Map', 'Manual'
    for test in tests:
        trace(test.ljust(max(map(len, tests)) + 1))
        match test:
            case 'In':
                trace(3 in X)                               # Membership
            case 'For':
                for i in X:                                  # for-loop iteration
                    trace(i, end='| ')
            case 'Comp':
                trace([i ** 2 for i in X])                  # Other Iteration tools
            case 'Map':
                trace(list(map(bin, X)))
            case 'Manual':
                I = iter(X)                                  # Manual iteration
                while True:
                    try:
                        trace(next(I), end='| ')
                    except StopIteration:
                        break
        print()

if __name__ == '__main__': self_test(Iter)                # Test Iters here

```

As is, the class in this file has an `__iter__` that supports only a single active scan at any point in time (e.g., nested loops won't work) because each iteration attempt resets the scan cursor to the front. Now that you know about `yield` in iteration methods, you should be able to tell that [Example 30-9](#) is equivalent but allows multiple active scans—and judge for yourself whether its more implicit nature is worth the nested-scan support (and 5 lines shaved).

Example 30-9. *contains-yield.py*

```

from contains import *

class ItersYield(Iter):
    def __iter__(self):                                    # Preferred for iteration
        trace('@iter @next')                               # Allows multiple active iterators
        for x in self.data:                                # Implicit generator alternative
            yield x
        trace('@next')

if __name__ == '__main__': self_test(IterYield)          # Test Iters here

```

When either version of this file runs, its output is as follows—the specific `__contains__` intercepts membership, the general `__iter__` catches other iteration tools such that `__next__` (whether explicitly coded or implied by `yield`) is called repeatedly, and `__getitem__` is never called:

```

$ python3 contains.py
In      @contains True
For      @iter @next 1 | @next 2 | @next 3 | @next 4 | @next
Comp     @iter @next @next @next @next @next [1, 4, 9, 16]

```

```

Map      @iter @next @next @next @next @next ['0b1', '0b10', '0b11', '0b100']
Manual   @iter @next 1 | @next 2 | @next 3 | @next 4 | @next

```

Watch, though, what happens to this code's output if we comment out its `__contains__` method—membership is now routed to the general `__iter__` instead (add triple quotes above and below the method to test live):

```

$ python3 contains.py
In      @iter @next @next @next True
For      @iter @next 1 | @next 2 | @next 3 | @next 4 | @next
Comp     @iter @next @next @next @next @next [1, 4, 9, 16]
Map      @iter @next @next @next @next @next ['0b1', '0b10', '0b11', '0b100']
Manual   @iter @next 1 | @next 2 | @next 3 | @next 4 | @next

```

And finally, here is the output if *both* `__contains__` and `__iter__` are commented out—the indexing `__getitem__` fallback is called with successively higher indexes until it raises `IndexError`, for membership and other iteration tools:

```

$ python3 contains.py
In      @get[0] @get[1] @get[2] True
For      @get[0] 1 | @get[1] 2 | @get[2] 3 | @get[3] 4 | @get[4]
Comp     @get[0] @get[1] @get[2] @get[3] @get[4] [1, 4, 9, 16]
Map      @get[0] @get[1] @get[2] @get[3] @get[4] ['0b1', '0b10', '0b11', '0b100']
Manual   @get[0] 1 | @get[1] 2 | @get[2] 3 | @get[3] 4 | @get[4]

```

As we've seen, the `__getitem__` method does other work too: besides iterations, it also intercepts explicit indexing as well as slicing. Slice expressions trigger `__getitem__` with a slice object containing bounds, both for built-in types and user-defined classes, so slicing is automatic in our class. With `__iter__` enabled or not:

```

$ python3
>>> from contains import Iters
>>> X = Iters('hack')
>>> X[0]                                     # Indexing: __getitem__(0)
@get[0] 'h'

>>> X[1:]                                   # Slicing: __getitem__(slice(...))
@get[slice(1, None, None)] 'ack'
>>> X[:-1]
@get[slice(None, -1, None)] 'hac'

```

Iterations, though, are more selective—we get the first of the following if `__iter__` is still commented out and the second if it's not (be sure to restart or reload after the file mod either way):

```

>>> list(X)
@get[0] @get[1] @get[2] @get[3] @get[4] ['h', 'a', 'c', 'k']

>>> list(X)
@iter @next @next @next @next @next ['h', 'a', 'c', 'k']

```

In more realistic iteration use cases that are not sequence-oriented, though, the `__iter__` method may be easier to write since it must not manage an integer index, and `__contains__` allows for membership optimization as a special case. While iteration is a rich topic, it's time to move on to the next stop on our overloading tour.

Attribute Access: `__getattr__` and `__setattr__`

In Python, classes can also intercept basic *attribute* access (a.k.a. qualification) when needed or useful. Specifically, for an *object* created from a class, the dot operator expression `object.attribute` can be implemented by your code too, for reference, assignment, and deletion contexts. We explored a limited example in this category which rerouted attribute fetches in the tutorial of [Chapter 28](#), but will review and expand on the topic here.

Attribute Reference

The `__getattr__` method intercepts attribute references. It's called with the attribute name as a string whenever you try to qualify an instance with an *undefined* (nonexistent) attribute name. It is *not* called if Python can find the attribute using its inheritance tree search procedure.

Because of its behavior, `__getattr__` is useful as a hook for responding to attribute requests in a generic fashion. It's commonly used to delegate calls to embedded (or “wrapped”) objects from a proxy controller object—of the sort introduced in [Chapter 28](#)'s introduction to *delegation*. This method can also be used to adapt classes to an interface or add *accessors* for data attributes after the fact—logic in a method that validates or computes an attribute after it's already being used with simple dot notation (possibly after a rename of the original).

The basic mechanism underlying these goals is straightforward—the following class catches attribute references, computing the value for one dynamically, and triggering an error for others unsupported with the `raise` statement described earlier in this chapter for iterators (and again, fully covered in [Part VII](#)):

```
>>> class Empty:
    def __getattr__(self, attrname):          # On self.undefined
        if attrname == 'age':
            return 40
        else:
            raise AttributeError(attrname)

>>> X = Empty()
>>> X.age                                     # Becomes X.__getattr__('age')
40
>>> X.name                                   # Unsupported attribute
...error text omitted...
AttributeError: name
```

Here, the `Empty` class and its instance `X` have no real attributes of their own, so the access to `X.age` gets routed to the `__getattr__` method; `self` is assigned the instance (`X`), and `attrname` is assigned the undefined attribute name string (`'age'`). The class makes `age` look like a real attribute by returning a real value as the result of the `X.age` qualification expression (`40`). In effect, `age` becomes a *dynamically computed* attribute—its value is formed by running code, not fetching an object.

For attributes that the class doesn't know how to handle, `__getattr__` raises the built-in `AttributeError` exception to tell Python that these are bona fide undefined names; asking for `X.name` triggers the error. You'll see `__getattr__` again when we explore delegation and properties at work in the next two chapters; let's move on to related tools here.

Attribute Assignment and Deletion

In the same department, the `__setattr__` intercepts *all* attribute assignments. If this method is defined or inherited, `self.attr = value` becomes `self.__setattr__('attr', value)`. Like `__getattr__`, this allows your class to catch attribute changes and validate or transform as desired.

This method is a bit trickier to use, though, because assigning to any `self` attributes within `__setattr__` calls `__setattr__` again, potentially causing an infinite *recursion loop* (and a fairly quick stack overflow exception!). In fact, this applies to all `self` attribute assignments anywhere in the class—all are routed to `__setattr__`, even those in other methods, and those to names other than that which may have triggered `__setattr__` in the first place. So be warned: this catches *all* attribute assignments.

If you wish to use this method, you can avoid loops by coding instance attribute assignments as assignments to attribute dictionary keys. That is, use `self.__dict__['name'] = x`, not `self.name = x`; because you're not assigning to `__dict__` itself, this avoids the loop:

```
>>> class Accesscontrol:
    def __setattr__(self, attr, value):
        if attr == 'age':
            self.__dict__[attr] = value + 10      # Not self.name=val or setattr
        else:
            raise AttributeError(attr + ' not allowed')

>>> X = Accesscontrol()
>>> X.age = 40                                # Becomes X.__setattr__('age', 40)
>>> X.age                                    # Found in __dict__ as usual
50
>>> X.name = 'Pat'                            # Unsupported attribute
...text omitted...
AttributeError: name not allowed
```

If you change the `__dict__` assignment within this class to either of the following, it triggers the infinite recursion loop and exception—both dot notation and its `setattr` built-in function equivalent (the assignment analog of `getattr`) fail when `age` is assigned outside the class:

```
self.age = value + 10                        # Loops!
setattr(self, attr, value + 10)              # Loops! (attr is 'age')
```

An assignment to any other `self` attribute within the class triggers a recursive `__setattr__` call too, though in this class ends less dramatically in the manual `AttributeError` exception:

```
self.other = 99                             # Recurs + fails, but doesn't loop
```

It's also possible to avoid recursive loops in a class that uses `__setattr__` by *rerouting* any attribute assignments to a higher superclass with a call, instead of assigning keys in `__dict__`:

```
object.__setattr__(self, attr, value + 10)    # OK: doesn't loop (preview)
```

This uses an explicit-class call to the implied object superclass above all topmost classes (and has a `super().__setattr__(...)` equivalent sans `self` per the sidebar “[The super Alternative](#)” on page 691). In fact, this alternative may be *required* in some classes per the upcoming note, though this is rare in practice.

A third attribute-management method, `__delattr__`, is passed the attribute name string and invoked on all attribute deletions (i.e., `del object.attr`). Like `__setattr__`, it must avoid recursive loops by running attribute deletions within the class using `__dict__`, or rerouting them to a superclass.



Attribute outliers: Preceding chapters mentioned that attributes coded with advanced class tools such as *slots* and *properties* are not physically stored in the instance’s `__dict__` namespace dictionary—and *slots* may even preclude a `__dict__` altogether. As noted, `dir` and `getattr` might be needed for listing and fetching attributes in classes using these tools, but assignment is similarly impacted: to support such “virtual” attributes, `__setattr__` may need to use the object.`__setattr__` scheme shown here, not `self.__dict__` indexing. You’ll learn much more about these attribute tools in upcoming chapters.

Other Attribute-Management Tools

The three attribute-access overloading methods we’ve met so far allow you to control or specialize access to attributes in your objects. They tend to play highly specialized roles, some of which we’ll explore later in this book. For another example of `__getattr__` at work, see [Chapter 28’s *person-composite.py* \(Example 28-11\)](#).

And for future reference, keep in mind that there are other ways to manage attribute access in Python:

- The `__getattribute__` method intercepts *all* attribute fetches, not just those that are undefined; like `__setattr__`, it must avoid loops for other attribute fetches in the class, usually with object rerouting.
- The property built-in function allows us to associate methods with fetch and set operations on a *specific* class attribute; it cannot catch accesses generically but can define what some do.
- *Descriptors* provide a protocol for associating `__get__` and `__set__` methods of a class with accesses to a *specific* instance or class attribute; they are as focused as property and, in fact, are used to implement it.
- *Slots* attributes are declared in classes but create implicit storage in each instance; if present, generic tools may need to list, fetch, and assign with schemes described in the preceding note.

Because these are all advanced tools that are not of interest to every Python programmer, we’ll defer the details of other attribute-management techniques until [Chapter 32](#), and await their focused coverage in [Chapter 38](#).

Emulating Privacy for Instance Attributes: Part 1

As another use case for attribute tools, the code in [Example 30-10](#)—file *private0.py*—generalizes the previous example, to allow each subclass to have its own list of private names that cannot be *assigned* to its instances (and raises a built-in exception with `raise`, which you’ll have to take on faith until [Part VII](#)).

Example 30-10. private0.py

```
class Privacy:
    def __setattr__(self, attr, value):          # On self.attr = value
        if attr in self.privates:
            raise NameError(f'{attr!r} for {self!r}')
        else:
            self.__dict__[attr] = value        # Avoid loops by using dict key

class Test1(Privacy):
```

```

privates = ['age']

class Test2(Privacy):
    privates = ['name', 'pay']
    def __init__(self):
        self.__dict__['name'] = 'Pat'           # To do better, see Chapter 39

if __name__ == '__main__':
    x = Test1()
    x.name = 'Sue'           # Works
    print(x.name)
    #x.age = 40              # Fails

    y = Test2()
    y.age = 30               # Works
    print(y.age)
    #y.name = 'Bob'         # Fails

```

This is a first-cut solution for an implementation of *attribute privacy* in Python—disallowing changes to attribute names outside a class. Although Python doesn’t support private declarations per se, techniques like this can emulate much of their purpose.

This particular attempt, though, is a partial—and even clumsy—solution. To make it more effective, we must augment it to allow classes to set their private attributes more naturally, without having to go through `__dict__` each time, as the constructor must do here to avoid triggering `__setattr__` and an exception. A better and more complete approach might require a wrapper (“proxy”) class to check for private attribute accesses made outside the class only and a `__getattr__` to validate attribute fetches too.

We’ll postpone a more complete solution to attribute privacy until [Chapter 39](#), where we’ll use *class decorators* to intercept and validate attributes more generally. Even though privacy can be emulated this way, though, it almost never is in practice. Python programmers are able to write large OOP frameworks and applications without private declarations—an interesting finding about access controls in general that is beyond the scope of our purposes here.

Still, catching attribute references and assignments is generally a useful technique; it supports *delegation*, a design technique that allows controller objects to wrap up embedded objects, add new behaviors, and route other operations back to the wrapped objects. Because they involve design topics, we’ll revisit delegation and wrapper classes in the next chapter. Here, it’s time to move ahead in the operator overloading domain.

String Representation: `__repr__` and `__str__`

Our next methods deal with display formats—a topic we’ve already explored in prior chapters but will summarize and formalize here. To serve as a guinea pig, the following codes the `__init__` constructor and the `__add__` overload method, both of which we’ve already seen (+ is an in-place operation here, just to show that it can be; this may be better coded as a named method per [Chapter 27](#), or the in-place `__iadd__` covered ahead). As we’ve learned, the default display of instance objects for a class like this is neither generally useful nor aesthetically pretty:

```

>>> class adder:
    def __init__(self, value=0):
        self.data = value           # Initialize data
    def __add__(self, other):
        self.data += other          # Add other in place

```

```

>>> x = adder()                                # Default displays:
>>> print(x)                                    # str or else repr
<__main__.adder object at 0x106110c80>
>>> x                                            # repr
<__main__.adder object at 0x106110c80>

```

But coding or inheriting string representation methods allows us to customize the display—as in the following, which defines a `__repr__` method in a subclass that returns a string representation for its instances:

```

>>> class addrepr(adder):                      # Inherit __init__, __add__
    def __repr__(self):                        # Add string representation
        return f'addrepr({self.data})'        # Convert to as-code string

>>> x = addrepr(2)
>>> x + 1
>>> x                                           # Runs __repr__
addrepr(3)
>>> print(x)                                   # Runs __repr__
addrepr(3)
>>> str(x), repr(x)                           # Runs __repr__ for both
('addrepr(3)', 'addrepr(3)')

```

If defined, `__repr__` (or its close relative, `__str__`) is called automatically when class instances are printed or converted to strings. These methods allow you to define a better display format for your objects than the default instance display. Here, `__repr__` uses basic string formatting to convert the managed `self.data` object to a more human-friendly string for display.

Why Two Display Methods?

So far, this section has largely been review. But while these methods are generally straightforward to use, their roles and behavior have some subtle implications for both design and coding. In particular, Python provides its two display methods to support alternative displays for different audiences:

- `__str__` is tried first for the `print` operation and the `str` built-in function (the internal equivalent of which `print` runs). It generally should return a *user-friendly* display.
- `__repr__` is used in all other contexts: for interactive echoes, the `repr` function, and nested appearances, as well as by `print` and `str` if no `__str__` is present. It should generally return an *as-code* string that could be used to re-create the object or a *detailed* display for developers.

That is, `__repr__` is used everywhere, except by `print` and `str` when a `__str__` is defined. This means you can code a `__repr__` to define a single display format used everywhere and may code a `__str__` to either support `print` and `str` exclusively or to provide an alternative display for them.

General tools may also prefer `__str__` to leave other classes the option of adding an alternative `__repr__` display for use in other contexts, as long as `print` and `str` displays suffice for the tool. Conversely, a general tool that codes a `__repr__` still leaves clients the option of adding alternative displays with a `__str__` for `print` and `str`. In other words, if you code either, the other is available for an additional display. In cases where the choice isn't clear, use `__str__` for higher-level displays and `__repr__` for lower-level displays and all-inclusive roles.

Let's write some code to illustrate these two methods' distinctions in more concrete terms. The prior example in this section showed how `__repr__` is used as the fallback option in many contexts. However,

while printing falls back on `__repr__` if no `__str__` is defined, the inverse is not true—other contexts, such as interactive echoes, use `__repr__` only and don't try `__str__` at all:

```
>>> class addstr(adder):
    def __str__(self):
        return f'[Value: {self.data}]'
# __str__ but no __repr__
# Convert to nice string

>>> x = addstr(3)
>>> x + 1
>>> x
# Default __repr__ (in object)
<__main__.addstr object at 0x106111b20>
>>> print(x)
# Runs __str__ (in addstr)
[Value: 4]
>>> str(x), repr(x)
('[Value: 4]', '<__main__.addstr object at 0x106111b20>')
```

Because of this, `__repr__` may be best if you want a *single* display for all contexts. By defining *both* methods, though, you can support different displays in different contexts—for example, a class's end-user display with `__str__`, and a class's developer display with `__repr__`. In effect, `__str__` simply overrides `__repr__` for more user-friendly display contexts:

```
>>> class addboth(adder):
    def __str__(self):
        return f'[Value: {self.data}]'
    def __repr__(self):
        return f'addboth({self.data})'
# User-friendly string
# As-code string

>>> x = addboth(4)
>>> x + 1
>>> x
# Runs __repr__
addboth(5)
>>> print(x)
# Runs __str__
[Value: 5]
>>> str(x), repr(x)
('[Value: 5]', 'addboth(5)')
```

Bonus: your classes' `__str__` and `__repr__` are also run automatically by all three *string-formatting* tools of [Chapter 7](#). This may not be a shocker, given these tools are defined to work like `str` and `repr` in this role, but display overloading is not just about REPL echoes and `print`:

```
>>> f'{x!s} {x!r}', '{!s} {!r}'.format(x, x), '%s %r' % (x, x)
('[Value: 5] addboth(5)', '[Value: 5] addboth(5)', '[Value: 5] addboth(5)')
```

Display Usage Notes

Though generally simple to use, three points regarding these display methods are worth calling out here. First, keep in mind that `__str__` and `__repr__` must both return *strings*; other result types are not converted and raise errors, so be sure to run them through a to-string converter (e.g., `str` or `f'...'`) if needed.

Second, depending on a container's string-conversion logic, the user-friendly display of `__str__` might only apply when objects appear at the top level of a print operation; objects *nested* in larger objects might still print with their `__repr__` or its default. The following illustrates both of these points:

```
>>> class Printer:
    def __init__(self, val):
        self.val = val
    def __str__(self):
        return str(self.val)

# Used for instance itself
# Convert to a string result

>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)

# __str__ run when instance printed
# But not when instance is in a list!
2
3
>>> print(objs)
[<__main__.Printer object at 0x106110c80>, <__main__.Printer object at 0x1060d2570>]
>>> objs
[<__main__.Printer object at 0x106110c80>, <__main__.Printer object at 0x1060d2570>]
```

To ensure that a custom display is run in all contexts regardless of the container, code `__repr__`, not `__str__`; the former is run in all cases if the latter doesn't apply, including nested appearances:

```
>>> class Printer:
    def __init__(self, val):
        self.val = val
    def __repr__(self):
        return str(self.val)

# __repr__ used by print if no __str__
# __repr__ used if echoed or nested

>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)

# No __str__: runs __repr__
2
3
>>> print(objs)

# Runs __repr__, not __str__
[2, 3]
>>> objs
[2, 3]
```

Third, and perhaps most subtle, the display methods also have the potential to trigger infinite *recursion loops* in rare contexts—because some objects' displays include displays of other objects, it's not impossible that a display may trigger a display of an object being displayed, and thus loop. This is rare and obscure enough to skip here but watch for an example of this looping potential to appear for these methods in a note near the end of the next chapter about its *listinherited.py* class of [Example 31-12](#), where `__repr__` can loop.

In practice, `__str__` and its more inclusive relative, `__repr__`, seem to be the second most commonly used operator-overloading methods in Python scripts, behind `__init__`. Anytime you can print an object and see a custom display, one of these two tools is probably in use. For additional examples of these tools at work and the design trade-offs they imply, see [Chapter 28](#)'s case study and [Chapter 31](#)'s class-lister mix-ins, as well as their role in [Chapter 35](#)'s exception classes, where `__str__` is required over `__repr__`.

Right-Side and In-Place Ops: `__radd__` and `__iadd__`

Our next group of overloading methods extends the functionality of binary operator methods such as `__add__` and `__sub__` (called for `+` and `-`), which we've already seen. As mentioned earlier, part of the reason there are so many operator-overloading methods is that they come in multiple flavors—for every binary expression, we can implement a *left*, *right*, and *in-place* variant. Though defaults are also applied if you don't code all three, your objects' roles dictate how many variants you'll need to code.

Right-Side Addition

For instance, the `__add__` methods coded so far technically do not support the use of instance objects on the *right* side of the `+` operator:

```
>>> class Adder:
    def __init__(self, value=0):
        self.data = value
    def __add__(self, other):
        return self.data + other

>>> x = Adder(5)
>>> x + 2
7
>>> 2 + x
TypeError: unsupported operand type(s) for +: 'int' and 'Adder'
```

To implement more general expressions and hence support *commutative*-style operators, code the `__radd__` method as well. Python calls `__radd__` only when the object on the right side of the `+` is your class instance, but the object on the left is not an instance of your class. The `__add__` method for the object on the left is called instead in all other cases. As a demo, consider the class in [Example 30-11](#) (which we'll be adding to in a moment).

Example 30-11. commuter.py (start)

```
class Commuter1:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other

    def __radd__(self, other):
        print('radd', self.val, other)
        return other + self.val
```

Because this defines both left- and right-side overloads for `+`, instances can appear on either side, or both:

```
>>> from commuter import Commuter1
>>> x = Commuter1(88)
>>> y = Commuter1(99)

>>> x + 1                                # __add__: instance + noninstance
add 88 1
89
>>> 1 + y                                # __radd__: noninstance + instance
radd 99 1
100
>>> x + y                                # __add__: instance + instance => triggers __radd__
add 88 <comuter.Commuter1 object at 0x1011b63f0>
radd 99 88
187
```

Notice how the order is reversed in `__radd__`: `self` is really on the right of the `+`, and `other` is on the left. Also note that `x` and `y` are instances of the same class here; when instances of different classes appear mixed in an expression, Python prefers the class of the one on the left. When we add the two

instances of this class together, Python runs `__add__`, which in turn triggers `__radd__` by simplifying the left operand and re-adding.

Reusing `__add__` in `__radd__`

For truly commutative operations that do not require special-casing by position, it is also sometimes sufficient to reuse `__add__` for `__radd__`, either by calling `__add__` directly; by swapping order and re-adding to trigger `__add__` indirectly; or by simply assigning `__radd__` to be an alias for `__add__` at the top level of the class statement (i.e., in the class's scope). The alternatives in [Example 30-12](#) implement all three of these schemes and return the same results as the original—though the last saves an extra call or dispatch and hence may be quicker (in all, `__radd__` is run when `self` is on the right side of a `+`).

Example 30-12. `commuter.py` (continued)

```
class Commuter2:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other

    def __radd__(self, other):
        return self.__add__(other)           # Call __add__ explicitly

class Commuter3:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other

    def __radd__(self, other):
        return self + other                 # Swap order and re-add

class Commuter4:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other

    __radd__ = __add__                      # Alias: cut out the middleperson
```

In all these, right-side instance appearances trigger the single, shared `__add__` method, passing the right operand to `self`, to be treated the same as a left-side appearance. Run these on your own for more insight; their names differ, but their usage and returned values are the same as the original `Commuter1` of [Example 30-11](#).

Propagating class type

In more realistic classes where the class type may need to be propagated in results, things can become trickier: type testing may be required to tell whether it's safe to convert and thus avoid nesting. For

instance, without the built-in `isinstance` test in [Example 30-13](#), we could wind up with a `Commuter5` whose `val` is another `Commuter5` when two instances are added and `__add__` triggers `__radd__`.

Example 30-13. `commuter.py` (continued)

```
class Commuter5:                                # Propagate class type in results
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        if isinstance(other, Commuter5):        # Type test to avoid object nesting
            other = other.val
        return Commuter5(self.val + other)       # Else + result is another Commuter

    def __radd__(self, other):
        return Commuter5(other + self.val)

    def __repr__(self):
        return f'Commuter5({self.val})'
```

When this version is run, `+` results retain the `Commuter5` type for future operations:

```
>>> from commuter import Commuter5
>>> x = Commuter5(88)
>>> y = Commuter5(99)

>>> x
Commuter5(88)
>>> x + 1                                # Result is another Commuter instance
Commuter5(89)
>>> 1 + y
Commuter5(100)

>>> z = x + y                            # Not nested: doesn't recur to __radd__
>>> z
Commuter5(187)
>>> z + 10
Commuter5(197)
>>> z + z
Commuter5(374)
>>> z + z + 1
Commuter5(375)
```

The need for the `isinstance` type test here is very subtle—uncomment, run, and trace to see why it's required. If you do, you'll see that the last part of the preceding test winds up differing and nesting objects—which still do the math correctly but kick off pointless recursive calls to simplify their values and extra constructor calls to build results:

```
>>> z = x + y                            # With isinstance test+action commented-out
>>> z
Commuter5(Commuter5(187))
>>> z + 10
Commuter5(Commuter5(197))
>>> z + z
Commuter5(Commuter5(Commuter5(Commuter5(374))))
>>> z + z + 1
Commuter5(Commuter5(Commuter5(Commuter5(375))))
```

Another way out of this dilemma is to test and simplify in the *constructor* instead, per [Example 30-14](#).

Example 30-14. *commuter.py* (continued)

```
class Commuter6:                                # Propagate class type in results
    def __init__(self, val):
        if isinstance(val, Commuter6):         # Type test to avoid object nesting
            self.val = val.val
        else:
            self.val = val

    def __add__(self, other):
        return Commuter6(self.val + other)

    def __radd__(self, other):
        return Commuter6(other + self.val)

    def __repr__(self):
        return f'Commuter6({self.val})'
```

This last version works the same as the non-nesting `Commuter5`. To test all of this section's classes, the rest of *commuter.py* in [Example 30-15](#) looks and runs like this—like functions, classes can again be used in tuples naturally because they are first-class objects.

Example 30-15. *commuter.py* (conclusion)

```
if __name__ == '__main__':
    for klass in (Commuter1, Commuter2, Commuter3, Commuter4, Commuter5, Commuter6):
        print('-' * 50)
        x = klass(88)
        y = klass(99)
        print(x + 1)
        print(1 + y)
        print(x + y)
```

\$ python3 commuter.py

```
-----
add 88 1
89
radd 99 1
100
add 88 <__main__.Commuter1 object at 0x101edc2f0>
radd 99 88
187
-----
...etc...
-----
Commuter6(89)
Commuter6(100)
Commuter6(187)
```

Experiment with these classes on your own for more insight. Aliasing `__radd__` to `__add__` in `Commuter5` and `Commuter6`, for example, works and saves a line but doesn't prevent object nesting without these classes' `isinstance` tests. See also Python's manuals for a discussion of other options in this domain; for example, classes may also return the special `NotImplemented` object for unsupported operands to influence method selection (this is treated as though the method were not defined—and differs from the prior chapter's `NotImplementedError`).

In-Place Addition

To also implement `+=` in-place augmented addition, code either an `__iadd__` or an `__add__`. The latter is used if the former is absent. In fact, the prior section's `Commuter` classes already support `+=` for this reason—Python runs `__add__` and assigns the result manually. The `__iadd__` method, though, allows for more efficient in-place changes to be coded where applicable; its return value is assigned to the target on the left of the `+=`:

```
>>> class Number:
    def __init__(self, val):
        self.val = val
    def __iadd__(self, other):
        self.val += other
        return self
# __iadd__ explicit: x += y
# Usually returns self
# Else None is returned+assigned

>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
```

For mutable objects, this method can often specialize for quicker in-place changes:

```
>>> y = Number([1])
>>> y += [2]
>>> y += [3]
>>> y.val
[1, 2, 3]
# In-place change faster than +
```

The normal `__add__` method is run as a fallback, but may not be able to optimize in-place cases:

```
>>> class Number:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        return Number(self.val + other)
# __add__ fallback: x = (x + y)
# Propagates class type

>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
# And += does concatenation here
```

Though we've focused on `+` here, keep in mind that *every* binary operator has similar right-side and in-place overloading methods that work the same (e.g., `__mul__`, `__rmul__`, and `__imul__`). Still, these methods tend to be uncommon in practice; you only code right-side methods for either-side roles and in-place methods for code economy or speed—and only if you need to support such operators at all. For instance, a `Vector` class may use these tools, but a `Button` class probably would not. `Button` presses, though, may run the next section's method.

Call Expressions: `__call__`

On to our next overloading method: the `__call__` method is called when your instance is called. No, this isn't a circular definition—if defined, Python runs a `__call__` method for function-call expressions applied to your instances, passing along whatever positional or keyword arguments were sent. This allows instances to conform to a function-based API:

```
>>> class Callee:
    def __call__(self, *pargs, **kargs):      # Intercept instance calls
        print(f'Called: {pargs=} {kargs=}')  # Accept arbitrary arguments

>>> C = Callee()
>>> C(1, 2, 3)                             # C is a "callable" object
Called: pargs=(1, 2, 3) kargs={}
>>> C(1, 2, 3, x=4, y=5)
Called: pargs=(1, 2, 3) kargs={'x': 4, 'y': 5}
```

More formally, all the argument-passing modes we explored in [Chapter 18](#) are supported by the `__call__` method—whatever is passed to the instance is passed to this method, along with the usual implied instance argument `self`. For example, the method definitions:

```
class C:
    def __call__(self, a, b, c=5, d=6): ...    # Normals and defaults

class C:
    def __call__(self, *pargs, **kargs): ...   # Collect arbitrary arguments

class C:
    def __call__(self, *pargs, d=6, **kargs): ... # 3.X keyword-only argument
```

all match all the following instance calls after assigning `X` to `C()`:

```
X(1, 2)                                # Omit defaults
X(1, 2, 3, 4)                          # Positionals
X(a=1, b=2, d=4)                        # Keywords
X(*[1, 2], **dict(c=3, d=4))            # Unpack arbitrary arguments
X(1, *(2,), c=3, **dict(d=4))           # Mixed modes
```

See [Chapter 18](#) for a refresher on function arguments. The net effect is that classes and instances with a `__call__` support the exact same argument syntax and semantics as normal functions and methods.

Intercepting call expression like this allows class instances to emulate the look and feel of things like functions, but also retain state information for use during calls. The following, for example, defines callable objects with per-call info specified by explicit attribute assignments:

```
>>> class Prod:
    def __init__(self, value):              # Accept just one argument
        self.value = value
    def __call__(self, other):
        return self.value * other

>>> x = Prod(2)                           # "Remembers" 2 in state
>>> x(3)                                  # 3 (passed) * 2 (state)
6
>>> x(4)
8
```

In this example, the `__call__` may seem a bit gratuitous at first glance. A simple method can provide similar utility:

```
>>> class Prod:
    def __init__(self, value):
        self.value = value
    def comp(self, other):
        return self.value * other

>>> x = Prod(3)
```

```
>>> x.comp(3)
9
```

However, `__call__` can become more beneficial when using APIs (i.e., libraries) that expect functions—it allows us to code objects that conform to an expected function-call interface but also retain state information and other class assets such as inheritance. In fact, it may be the third most commonly used operator-overloading method, behind the `__init__` constructor and the `__str__` and `__repr__` display-format alternatives (qualitatively speaking).

Function Interfaces and Callback-Based Code

As an example, Python’s `tkinter` GUI module lets you register functions as event handlers (a.k.a. *callbacks*)—when events occur, `tkinter` calls the registered objects. If you want an event handler to retain state between events, you can register a class *instance* that conforms to the expected interface with `__call__`. [Chapter 17’s closure functions](#) can achieve similar effects but don’t provide as much support for multiple operations or customization.

To demo the concept, here’s a hypothetical example of `__call__` applied to the GUI domain. The following class defines an object that supports a function-call interface but also has state information that remembers the color a button should change to when it is later pressed:

```
class Callback:
    def __init__(self, color):           # Function + state information
        self.color = color
    def __call__(self):                 # Support calls with no arguments
        print('turn', self.color)
```

In the context of a GUI, we can register instances of this class as event handlers for buttons, even though the GUI expects to be able to invoke event handlers as simple functions with no arguments:

```
cb1 = Callback('blue')                 # Remember blue
cb2 = Callback('green')                # Remember green

B1 = Button(command=cb1)                # Register handlers
B2 = Button(command=cb2)
```

When the button is later pressed, the instance object is called as a simple function with no arguments, exactly like in the following calls. Because it retains state as instance attributes, though, it remembers what to do—it becomes a *stateful function* object:

```
cb1()                                  # On event: prints 'turn blue'
cb2()                                  # On event: prints 'turn green'
```

In fact, some consider such classes to be the best way to retain state information in the Python language. With OOP, the state remembered is made explicit with attribute assignments. This is different than other state retention techniques (e.g., global variables, enclosing function scope references, and default mutable arguments), which rely on more limited or implicit behavior. Moreover, the added structure and customization in classes goes beyond state retention.

On the other hand, tools such as closure functions are useful in basic state retention roles too, and the `nonlocal` statement makes enclosing scopes a viable alternative in more programs. We’ll revisit such trade-offs when we start coding substantial decorators in [Chapter 39](#), but here’s a quick *closure* equivalent:

```
def callback(color):                                # Enclosing scope versus attrs
    def oncall():
        print('turn', color)
    return oncall

cb3 = callback('yellow')                            # Handler to be registered
cb3()                                                # On event: prints 'turn yellow'
```

Before we move on, there are two other ways that Python programmers sometimes tie information to a callback function like this. One option is to use default arguments in `lambda` functions:

```
cb4 = (lambda color='red':                          # Defaults retain state too
       print('turn', color))                       # lambda defers code till call
cb4()                                                # On event: prints 'turn red'
```

The other is to use *bound methods* of a class—covered in the next chapter but simple enough to preview here. A bound-method object is created for instance methods referenced but not called and remembers both the instance and the referenced function. This object may therefore be called later as a simple function without an instance:

```
class Callback:
    def __init__(self, color):                      # Class with state information
        self.color = color
    def changeColor(self):                          # A normally named method
        print('turn', self.color)

cb1 = Callback('blue')
cb2 = Callback('yellow')

B1 = Button(command=cb1.changeColor)               # Bound method: reference, not call
B2 = Button(command=cb2.changeColor)               # Remembers instance + function pair
```

In this case, when this button is later pressed it's as if the GUI does the following, which invokes the instance's `changeColor` method to process the `cb1` object's state information instead of calling the instance itself:

```
cb1 = Callback('blue')
cmd = cb1.changeColor                              # Registered event handler
cmd()                                                # On event: prints 'turn blue'
```

Note that a `lambda` is not required here unless extra arguments must be passed, because a bound method reference by itself already *defers* a call until later. This technique doesn't require overloading calls with `__call__`, but either scheme may be preferred for a given program. Again, watch for more about bound methods in the next chapter.

You'll also see another `__call__` example in [Chapter 32](#), where we will use it to implement a function decorator—a callable object often used to add a layer of logic on top of an embedded function. Because `__call__` allows us to attach state information to a callable object, it's a natural implementation technique for a function that must remember to call another function when called itself. For more `__call__` examples, also watch for the more advanced decorators and metaclasses of [Chapters 39](#) and [40](#).

Comparisons: `__lt__`, `__gt__`, and Others

Our next batch of overloading methods supports object comparisons. As suggested in [Table 30-1](#), classes can define methods to catch all six comparison operators: `<`, `>`, `<=`, `>=`, `=`, and `!=`. These methods are generally straightforward to use, but keep the following qualifications in mind:

- Unlike the `__add__`/`__radd__` pairings discussed earlier, there are no *right-side* variants of comparison methods. Instead, reflective methods are used when only one operand supports comparison (e.g., `__lt__` and `__gt__` are each other's reflection).
- There are no implicit *relationships* among the comparison operators. The truth of `==` does not imply that `!=` is false, for example, so both `__eq__` and `__ne__` should be defined to ensure that both operators behave correctly.

We don't have space for an in-depth exploration of comparison methods, but as a quick introduction, consider the following class and tests:

```
>>> class Vetter:
    data = 'hack'
    def __gt__(self, other):
        print(f'gt: {self=} {other=}')
        return self.data > other
    def __lt__(self, other):
        print(f'lt: {self=} {other=}')
        return self.data < other

>>> X = Vetter()
>>> X > 'code', X < 'code'
gt: self=<__main__.Vetter object at 0x10898f650> other='code'
lt: self=<__main__.Vetter object at 0x10898f650> other='code'
(True, False)

>>> 'code' < X, 'code' > X
gt: self=<__main__.Vetter object at 0x10898f650> other='code'
lt: self=<__main__.Vetter object at 0x10898f650> other='code'
(True, False)

>>> X < X
lt: self=<__main__.Vetter object at 0x10898f650> other=<__main__.Vetter ...etc...>
gt: self=<__main__.Vetter object at 0x10898f650> other='hack'
False
```

When run, the class's methods intercept and implement comparison expressions as noted by their trace outputs. Importantly, `__lt__` is also used for *sorts*—both the list method and built-in function:

```
>>> class Order:
    def __init__(self, data):
        self.data = data
    def __lt__(self, other):
        return self.data < other.data
    def __repr__(self):
        return f'Order({self.data})'

>>> sorted(Order(i) for i in [3, 1, 4, 2])
[Order(1), Order(2), Order(3), Order(4)]
```

Consult Python's manuals for more details in this category. As you'll find there, the `__eq__` method run for value equality is coupled with the `__hash__` method run for as-key and set-object roles (in ways that should send most readers screaming into the night); and comparison methods can also return `NotImplemented` for unsupported arguments (but again, not `NotImplementedError`, an exception with a similar name but very different roles).

Boolean Tests: `__bool__` and `__len__`

The next set of methods is truly useful (pun intended). As you’ve learned, every object is inherently true or false in Python. When you code classes, you can define what this means for your objects by coding methods that give the True or False values of instances on request.

In Boolean contexts, Python first tries `__bool__` to obtain a direct Boolean value; if that method is missing, Python tries `__len__` to infer a truth value from the object’s length—a length of zero means empty, which is always false. The first of these generally uses object state or other information to produce a Boolean result:

```
>>> class Truth:
    def __bool__(self): return True

>>> X = Truth()
>>> if X: print('yes!')

yes!

>>> class Truth:
    def __bool__(self): return False

>>> X = Truth()
>>> bool(X)
False
```

If this method is missing, Python falls back on length because a nonempty object is considered true. That is, a nonzero length is taken to mean the object is true, and a zero length means it is false—just as for built-in objects:

```
>>> class Truth:
    def __len__(self): return 0           # Empty means false too

>>> X = Truth()
>>> if not X: print('no!')

no!
```

If *both* methods are present Python prefers `__bool__` over `__len__`, because it is more specific:

```
>>> class Truth:
    def __bool__(self): return True       # Preferred over length
    def __len__(self): return 0          # Object length: fallback

>>> if Truth(): print('yes!')

yes!
```

If *neither* truth method is defined, the object is vacuously considered true (though any existential implications of this are strictly out of scope here):

```
>>> class Truth:
    pass

>>> X = Truth()
>>> bool(X)
True
```

But now that we’ve managed to cross over into the realm of philosophy, let’s move on to look at one last overloading context: *object demise*.

Object Destruction: `__del__`

It's time to close out this chapter—and learn how to do the same for our class objects. You've seen how the `__init__` *constructor* is called whenever an instance is generated (and noted how `__new__` is run first to make the object). Its counterpart, the *destructor* (less commonly known as *finalizer*) method `__del__`, is run automatically when an instance's space is being reclaimed (i.e., at garbage-collection time):

```
>>> class Life:
    def __init__(self, name):
        print('Hello', name)
        self.name = name
    def live(self):
        print(self.name + '...')
    def __del__(self):
        print('Goodbye', self.name)

>>> pat = Life('Pat')
Hello Pat
>>> pat.live()
Pat...
>>> pat = 'end'
Goodbye Pat
```

Here, when `pat` is assigned a string at the end, we lose the last reference to the `Life` instance and so trigger its destructor method. This works, and it may be useful for implementing some cleanup activities, such as terminating a server connection. However, destructors are not as commonly used in Python as in some OOP languages, for a number of reasons that the next section describes.

Destructor Usage Notes

The destructor method works as documented, but it has some well-known caveats and a few outright dark corners that make it somewhat rare to see in Python code:

Need

For one thing, destructors may not be as useful in Python as they are in some other OOP languages. Because Python automatically reclaims all *memory space* held by an instance when the instance is reclaimed, destructors are not necessary for space management. In the current CPython implementation of Python, you also don't need to close *file objects* held by the instance in destructors because they are automatically closed when reclaimed. As mentioned in [Chapter 9](#), though, it's still sometimes best to run file close methods anyhow because this autoclose behavior may vary in alternative Python implementations.

Predictability

For another, you cannot always easily predict when an instance will be reclaimed. In some cases, there may be lingering references to your objects in system tables that prevent destructors from running when your program expects them to be triggered. Python also does not guarantee that destructor methods will be called for objects that still exist when the interpreter exits.

Exceptions

In fact, `__del__` can be tricky to use for even more subtle reasons. Exceptions raised within it, for example, simply print a warning message to `sys.stderr` (the standard error stream) rather than triggering an exception event, because of the unpredictable context under which it is run by the garbage collector—it's not always possible to know where such an exception should be delivered.

Cycles

In addition, cyclic (a.k.a. circular) references among objects may prevent garbage collection from happening when you expect it to. An optional cycle detector, enabled by default, can automatically collect such objects eventually (including objects with `__del__` methods, as of Python 3.4). Since this is relatively obscure, we'll ignore further details here; see Python's standard manuals' coverage of both `__del__` and the `gc` garbage collector module for more information.

Because of these downsides, it's often better to code termination activities in an explicitly called method (e.g., `shutdown`). As described in the next part of the book, the `try/finally` statement also supports termination actions, as does the `with` statement for objects that support its context-manager model.

Chapter Summary

That's as many overloading examples as we have space for here. Most of the other operator-overloading methods work similarly to the ones we've explored, and all are just hooks for intercepting built-in type operations. Some overloading methods, for example, have unique argument lists or return values, but the general usage pattern is the same. You'll see a few others in action later in the book:

- **Chapter 34** uses `__enter__` and `__exit__` in `with` statement context managers.
- **Chapter 38** uses the `__get__` and `__set__` class descriptor `fetch/set` methods.
- **Chapter 40** uses the `__new__` object creation method in the context of metaclasses.

In addition, some of the methods we've studied here, such as `__call__` and `__str__`, will be employed by later examples in this book. For complete coverage, though, it must defer to other documentation sources—see Python's language manual or other reference resources for details on additional overloading methods.

In the next chapter, we leave the realm of class mechanics behind to explore *design*—the ways that classes are commonly used and combined to optimize code reuse. After that, we'll survey a gumbo of advanced class topics and move on to exceptions, the last core subject of this book. Before you read on, though, take a moment to work through the chapter quiz below to review the concepts we've covered here.

Test Your Knowledge: Quiz

1. What two operator-overloading methods can you use to support iteration in your classes?
2. What two operator-overloading methods handle printing, and in what contexts?
3. How can you intercept slice operations in a class?
4. How can you catch in-place addition in a class?
5. When should you provide operator overloading?

Test Your Knowledge: Answers

1. Classes can support iteration by defining (or inheriting) `__getitem__` or `__iter__`. In all iteration tools, Python tries to use `__iter__` first, which returns an object that supports the iteration protocol with a `__next__` method: if no `__iter__` is found by inheritance search, Python falls back on the `__getitem__` indexing method, which is called repeatedly, with successively higher indexes. If used, the `yield` statement can create the `__next__` method automatically.
2. The `__str__` and `__repr__` methods implement object print displays. The former is called by the `print` and `str` built-in functions; the latter is called by `print` and `str` if there is no `__str__`, and always by the `repr` built-in, interactive echoes, and nested appearances. That is, `__repr__` is used everywhere, except by `print` and `str` when a `__str__` is defined. A `__str__` is usually used for user-friendly displays; `__repr__` gives extra details or the object's as-code form. String formatting runs these methods too.
3. Slicing is caught by the `__getitem__` indexing method: it is called with a `slice` object instead of a simple integer index, and slice objects may be passed on or inspected as needed.
4. In-place addition tries `__iadd__` first, and `__add__` with an assignment second. The same policy holds true for all binary operators. The `__radd__` method is also available for right-side addition.
5. When a class naturally matches, or needs to emulate, a built-in type's interfaces. For example, collections might imitate sequence or mapping interfaces, and callables might be coded for use with an API that expects a function. You generally shouldn't implement expression operators if they don't naturally map to your objects naturally and logically, though—use normally named methods instead.

Designing with Classes

So far in this part of the book, we’ve concentrated on using Python’s OOP tool, the *class*. But OOP is also about *design*—that is, how to use classes to model useful objects. Toward this end, this chapter codes common OOP design patterns in Python, such as inheritance, composition, delegation, and factories. Along the way, we’ll also investigate some design-focused class concepts, such as pseudoprivate attributes, multiple inheritance, and bound methods. Because multiple inheritance is dependent on the MRO search order, we’ll finally explore that here too.

One note up front: some of the design terms mentioned here require more coverage than this book can provide. If this material sparks your curiosity, you may want to consider exploring a text on OOP design or design patterns as a next step. As you’ll see, the good news is that Python makes many traditional design patterns almost trivial.

Python and OOP

Let’s begin with a review—Python’s implementation of OOP can be summarized by three ideas:

Inheritance

Inheritance is based on attribute lookup in Python (in `X.name` expressions).

Polymorphism

In `X.method`, the meaning of `method` depends on the type (class) of subject object `X`.

Encapsulation

Methods and operators implement behavior, though data hiding is a convention by default.

By now, you should have a good feel for what basic *inheritance* is all about in Python, so we’ll take it as a given here. As you’ve learned, it’s the mechanism behind flexible code customization.

We’ve also talked about Python’s *polymorphism* a few times already. It flows from Python’s lack of type declarations (per [Chapter 6](#), the optional, unused, and paradoxical “type hinting” doesn’t qualify). Because attributes are always resolved at runtime, objects that implement the same interfaces are automatically interchangeable; clients don’t need to know what sorts of objects are implementing the methods they call.

Newer here, *encapsulation* in Python means *packaging*—that is, hiding implementation details behind an object’s interface. It does not mean enforced privacy, though that can be partly implemented with

code, as you’ll see in [Chapter 39](#). Encapsulation is available and useful in Python nonetheless: it allows the implementation of an object’s interface to be changed without impacting the users of that object.

Polymorphism Means Interfaces, Not Call Signatures

Some OOP languages also define polymorphism to mean overloading functions based on the type signatures of their *arguments*—the number passed and/or their types. Because there are no real type declarations in Python, this concept doesn’t apply; as we’ve seen, polymorphism in Python is based on object *interfaces*, not types.

If you’re pining for your C++ days, you can try to overload methods by their argument lists, like this:

```
class C:
    def meth(self, x):
        ...
    def meth(self, x, y, z):
        ...
```

Such code will run without error, but because the `def` simply assigns an object to a name in the class’s scope, the *last* definition of the method function is the only one that will be retained. Put another way, it’s just as if you say `X = 1` and then `X = 2`; at the end, `X` will be 2. Hence, there can be only one definition of a method name. Per [Chapter 29](#), this includes `__init__` constructors, which are only special because of when they are run.

If call-signature dispatch is truly required, you can always code type-based selections using the type-testing ideas we met in [Chapters 4](#) and [9](#), or the argument-list tools introduced in [Chapter 18](#):

```
class C:
    def meth(self, *args):
        if len(args) == 1:                # Branch on number arguments
            ...
        elif type(arg[0]) == int:         # Branch on argument types (or isinstance())
            ...
```

You normally shouldn’t do this, though—it’s not the “Python way.” As explained in [Chapter 16](#), you should write your code to expect only an object *interface*, not a specific object *type*. That way, it will be useful for a broader category of types and applications, both now and in the future:

```
class C:
    def meth(self, x):
        x.operation()                    # Assume x does the right thing
```

It’s also generally considered better to use distinct method *names* for distinct operations rather than relying on call signatures (no matter what language you code in).

But enough review. Although Python’s object model is straightforward, much of the art in OOP is in the way we combine classes to achieve a program’s goals. The next section begins a tour of some of the ways larger programs use classes to their advantage.

OOP and Inheritance: “Is-a” Relationships

We’ve explored the mechanics of inheritance in depth already; let’s turn to an example of how it can be used to model real-world relationships. From a *programmer’s* point of view, inheritance is kicked off by attribute qualifications, which trigger searches for names in instances, their classes, and then any super-classes. From a *designer’s* point of view, inheritance is a way to specify set membership: a class defines a set of properties that may be inherited and customized by more specific sets (i.e., subclasses).

To illustrate, let's put that pizza-making robot we talked about at the start of this part of the book to work. Suppose we've decided to explore alternative career paths and open a pizza restaurant (not bad, as career paths go). One of the first things we'll need to do is hire employees to serve customers, prepare the food, and so on. Being engineers at heart, we've decided to build a robot to make the pizzas, but being politically and cybernetically correct, we've also decided to make our robot a full-fledged employee with a salary.

Our pizza shop team can be simulated by the four classes in [Example 31-1](#), *employees.py*. The most general class, `Employee`, is a takeoff on [Chapter 28](#)'s demo. It provides common behavior such as bumping up salaries (`giveRaise`) and printing (`__repr__`). There are two kinds of employees, and so there are two subclasses of `Employee`—`Chef` and `Server`. Both override the inherited `work` method to print more specific messages. Finally, our pizza robot is modeled by an even more specific class—`PizzaRobot` is a kind of `Chef`, which is a kind of `Employee`. In OOP terms, we call these relationships “is-a” links: a robot is a chef, which is an employee.

Example 31-1. employees.py

```
class Employee:
    def __init__(self, name, salary=0):
        self.name = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary += self.salary * percent
    def work(self):
        print(self.name, 'does stuff')
    def __repr__(self):
        return (f'<{self.__class__.__name__}: '
                f'f{name="{self.name}", salary={self.salary:,.2f}}>')
```

```
class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print(self.name, 'makes food')
```

```
class Server(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print(self.name, 'interfaces with customer')
```

```
class PizzaRobot(Chef):
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print(self.name, 'makes pizza')
```

```
if __name__ == '__main__':
    pat = PizzaRobot('pat')          # Make a robot named pat
    print(pat)                       # Run inherited __repr__
    pat.work()                       # Run type-specific action
    pat.giveRaise(0.20)              # Give pat a 20% raise
    print(pat); print()

    for klass in Employee, Chef, Server, PizzaRobot:
        object = klass(klass.__name__)
        object.work()
```

When we run the self-test code included in this module, we create a pizza-making robot named `pat`, which inherits names from three classes: `PizzaRobot`, `Chef`, and `Employee`. For instance, printing and giving a raise to `pat` runs the `Employee` class's `__repr__` and `giveRaise` methods two levels up, respectively, simply because that's where the inheritance search finds these methods:

```
$ python3 employees.py
<PizzaRobot: name="pat", salary=50,000.00>
pat makes pizza
<PizzaRobot: name="pat", salary=60,000.00>

Employee does stuff
Chef makes food
Server interfaces with customer
PizzaRobot makes pizza
```

In a class hierarchy like this, you can usually make instances of any of the classes, not just the ones at the bottom. For instance, the `for` loop in this module's self-test code creates instances of all four classes; each responds differently when asked to work because the `work` method is different in each. `pat` the robot, for example, gets work from the most specific (i.e., lowest) `PizzaRobot` class.

Of course, these classes just *simulate* real-world objects; `work` prints a message for the time being, but it could be expanded to do real work later (see Python's interfaces to devices such as serial ports, Arduino boards, and the Raspberry Pi if you're taking this section much too literally!).



The super re-reminder: Notice how [Example 31-1](#) uses explicit class calls to run superclass constructors; this is how `PizzaRobot` salaries are set. Per the sidebar “[The super Alternative](#)” on [page 691](#), these could also use the `super().__init__(...)` form explored in the next chapter. As you'll find, this call avoids having to pass `self` along in single-inheritance trees like those here, but it's much more complex in the multiple-inheritance contexts you'll meet ahead. If you're anxious to see what this looks like now, see the alternative `employees-super.py` in the examples package.

OOP and Composition: “Has-a” Relationships

The notion of composition was introduced in [Chapters 26](#) and [28](#). From a *programmer's* perspective, composition involves embedding other objects in a container object and activating them to implement container methods. To a *designer*, composition is another way to represent relationships in a problem domain. But, rather than set membership, composition has to do with components—parts of a whole.

Composition also reflects the relationships between parts, called “*has-a*” relationships. Some OOP design texts refer to composition as *aggregation* or distinguish between the two terms by using aggregation to describe a weaker dependency between container and contained. In this text, a “composition” simply refers to a collection of embedded objects. The composite class generally provides an interface all its own and implements it by directing the embedded objects.

Now that we've implemented our employees, let's put them in the pizza shop and let them get busy. Our pizza shop is a composite object: it has an oven, and it has employees like servers and chefs. When a customer enters and places an order, the components of the shop spring into action—the server takes the order, the chef makes the pizza, and so on. [Example 31-2](#)—file `pizzashop.py`—simulates all the objects and relationships in this scenario.

Example 31-2. pizzashop.py

```
from employees import PizzaRobot, Server    # From Example 31-1

class Customer:
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print(self.name, 'orders from', server)
    def pay(self, server):
        print(self.name, 'pays for item to', server)

class Oven:
    def bake(self):
        print('oven bakes')

class PizzaShop:
    def __init__(self):
        self.server = Server('Jan')           # Embed other objects
        self.chef   = PizzaRobot('Pat')       # A robot named Pat
        self.oven   = Oven()

    def order(self, name):
        customer = Customer(name)             # Activate other objects
        customer.order(self.server)           # Customer orders from server
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)

if __name__ == '__main__':
    scene = PizzaShop()                       # Make the composite
    scene.order('Sue')                         # Simulate Sue's order
    print('...')
    scene.order('Bob')                         # Simulate Bob's order
```

The `PizzaShop` class is a container and controller; its constructor makes and embeds instances of the employee classes we wrote in the prior section, as well as an `Oven` class defined here. When this module's self-test code calls the `PizzaShop`'s `order` method, the embedded objects are asked to carry out their actions in turn.

Notice that we make a new `Customer` object for each order, and we pass on the embedded `Server` object to `Customer` methods; customers come and go, but the server is part of the pizza shop composite. Also notice that employees are still involved in an inheritance relationship; composition and inheritance are complementary tools.

When we run this module, our pizza shop handles two orders—one from Sue and then one from Bob (overlapping orders with the async coroutines of [Chapter 20](#) is explicitly out of scope here):

```
$ python3 pizzashop.py
Sue orders from <Server: name="Jan", salary=40,000.00>
Pat makes pizza
oven bakes
Sue pays for item to <Server: name="Jan", salary=40,000.00>
...
Bob orders from <Server: name="Jan", salary=40,000.00>
Pat makes pizza
oven bakes
Bob pays for item to <Server: name="Jan", salary=40,000.00>
```

Again, this is mostly just a toy simulation, but the objects and interactions are representative of composites at work. As a rule of thumb, classes can represent just about any objects and relationships you can express in a sentence; just replace *nouns* with classes (e.g., Oven) and *verbs* with methods (e.g., bake), and you'll have a first cut at a design.

Stream Processors Revisited

For a composition example that may be a bit more tangible than pizza-making robots, recall the generic data-stream processor function we partially coded in the introduction to OOP in [Chapter 26](#), repeated here for ease:

```
def processor(reader, converter, writer):
    while True:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Rather than using a simple function here, we might code this as a class that uses composition to do its work in order to provide more structure and support inheritance. [Example 31-3](#), file *streams.py*, demonstrates one way to code the class (it also mutates one method name, `readline`, because we're actually going to run this code here).

Example 31-3. *streams.py*

```
class Processor:
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer

    def process(self):
        while True:
            data = self.reader.readline()
            if not data: break
            data = self.converter(data)
            self.writer.write(data)

    def converter(self, data):
        assert False, 'converter must be defined'          # Or raise exception
```

This class defines a `converter` method that it expects subclasses to fill in; it's an example of the *abstract superclass* model we outlined in [Chapter 29](#) (again, more on `assert` in [Part VII](#)—it simply raises an exception if its test is false). Coded this way, `reader` and `writer` objects are embedded within the class instance (*composition*), and we supply the conversion logic in a subclass rather than passing in a converter function (*inheritance*). The file in [Example 31-4](#), *converters.py*, shows how.

Example 31-4. *converters.py*

```
from streams import Processor

class Uppercase(Processor):
    def converter(self, data):
        return data.upper()

if __name__ == '__main__':
    import sys
```

```
obj = Uppercase(open('trihack.txt'), sys.stdout)
obj.process()
```

Here, the `Uppercase` class inherits the stream-processing loop logic of `process` (and anything else that may be coded in its superclasses). It needs to define only what is unique about it—the data conversion logic. When this file is run, it makes and runs an instance that reads from the file *trihack.txt* in the current directory and writes the uppercase equivalent of that file to the `stdout` stream (which usually means the window you’re working in):

```
$ cat trihack.txt           # Use "type" on Windows
hack
Hack
HACK!

$ python3 converters.py
HACK
HACK
HACK!
```

To process different sorts of streams, pass in different sorts of objects to the class construction call. Here, we use an output file instead of a stream:

```
$ python3
>>> import converters
>>> scan = converters.Uppercase(open('trihack.txt'), open('trihackup.txt', 'w'))
>>> scan.process()

$ cat trihackup.txt
HACK
HACK
HACK!
```

But, as suggested earlier, we could also pass in arbitrary objects coded as classes that define the required input and output method interfaces. Here’s a simple example that passes in a writer class that wraps up the text inside HTML tags—lines are read from a file, run through uppercase conversion, and then printed with HTML tags:

```
$ python3
>>> from converters import Uppercase
>>> class HTMLize:
>>>     def write(self, line):
>>>         print(f'<PRE>{line.rstrip()}</PRE>')

>>> Uppercase(open('trihack.txt'), HTMLize()).process()
<PRE>HACK</PRE>
<PRE>HACK</PRE>
<PRE>HACK!</PRE>
```

If you trace through this example’s control flow, you’ll see that we get *both* uppercase conversion (by inheritance) and HTML formatting (by composition), even though the core processing logic in the original `Processor` superclass knows nothing about either step. The processing code only cares that writers have a `write` method and that a method named `converter` is defined; it doesn’t care what those methods do when they are called. Such polymorphism and encapsulation of logic are behind much of the power of classes in Python.

As is, the `Processor` superclass only provides a file-scanning loop. In more realistic work, we might extend it to support additional programming tools for its subclasses and, in the process, turn it into a full-blown application *framework*. Coding such a tool once in a superclass enables you to reuse it in all

your programs. Even in this simple example, because so much is packaged and inherited with classes, all we had to code was the HTML formatting step; the rest was free.

For another example of composition at work, see exercise 9 in “[Test Your Knowledge: Part VI Exercises](#)” on page 866 and its solution in [Appendix B](#); it’s similar to the pizza shop example. We’ve focused on inheritance in this book because that is the main tool that the Python language itself provides for OOP. But, in practice, composition may be used as much as inheritance as a way to structure classes, especially in larger systems. As we’ve seen, inheritance and composition are often complementary (and sometimes alternative) techniques. Because composition is a design issue outside the scope of the Python language and this book, though, we’ll defer to other resources for more on this topic.

Why You Will Care: Classes and Persistence

We’ve explored Python’s `pickle` and `shelve` object persistence earlier in this part of the book because it works especially well with class instances. In fact, these tools are often compelling enough to motivate the use of classes in general—by pickling or shelving a class instance, we store both data and logic.

For example, besides allowing us to simulate real-world interactions, the pizza shop classes developed in this chapter could also be used as the basis of a restaurant database. Pickling instances of such classes to a file makes them persistent across Python program executions:

```
>>> from pizzashop import PizzaShop
>>> shop = PizzaShop()
>>> shop.chef
<PizzaRobot: name="Pat", salary=50,000.00>
>>> import pickle
>>> pickle.dump(shop, open('shopfile.pkl', 'wb'))
```

This stores an entire composite shop object in a file all at once. To bring it back later in another session or program, a single step suffices as well. Objects restored this way retain both state and behavior:

```
>>> import pickle
>>> shop = pickle.load(open('shopfile.pkl', 'rb'))
>>> shop.chef
<PizzaRobot: name="Pat", salary=50,000.00>
>>> shop.order('sue')
sue orders from <Server: name="Jan", salary=40,000.00>
Pat makes pizza
oven bakes
sue pays for item to <Server: name="Jan", salary=40,000.00>
```

This is just a prototype as is, but we might extend the shop to keep track of inventory, revenue, and so on—saving it to its file after changes would retain its updated state. See the standard-library manual and related coverage in [Chapters 9](#), [28](#), and [37](#) for more on pickles and shelves.

OOP and Delegation: “Like-a” Relationships

Besides inheritance and composition, object-oriented programmers often speak of *delegation*, which usually implies controller objects that embed other objects to which they pass off operation requests. The controllers can take care of administrative activities, such as logging or validating accesses, adding extra steps to interface components, or monitoring active instances.

In a sense, delegation is a special form of composition, with a single embedded object managed by a *proxy* (sometimes called a *wrapper*) class that retains most or all of the embedded object's interface. The notion of proxies sometimes applies to other mechanisms, too, such as function calls; in delegation, we're concerned with proxies for *all* of an object's behavior, including method calls and other operations.

This concept was introduced by example in [Chapter 28](#), and in Python is often implemented with the `__getattr__` method hook we studied in [Chapter 30](#). Because this operator-overloading method intercepts accesses to nonexistent attributes, a wrapper class can use `__getattr__` to route arbitrary accesses to a wrapped object. Because this method allows attribute requests to be routed generically, the wrapper class retains the interface of the wrapped object and may add additional operations of its own.

By way of review, consider the file *trace.py* in [Example 31-5](#).

Example 31-5. *trace.py*

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object          # Save object
    def __getattr__(self, attrname):
        print('Trace: ' + attrname)    # Trace fetch
        return getattr(self.wrapped, attrname) # Delegate fetch
```

Recall from [Chapter 30](#) that `__getattr__` gets the attribute name as a string. This code makes use of the `getattr` built-in function to fetch an attribute from the wrapped object by name string—`getattr(X, N)` is like `X.N`, except that `N` is an *expression* that evaluates to a string at runtime, not a variable. In fact, `getattr(X, N)` is similar to `X.__dict__[N]`, but the former also performs an inheritance search, like `X.N`, while the latter does not (see [Chapters 23](#) and [29](#) for more on the `__dict__` attribute).

You can use the approach of this module's wrapper class to manage access to any object with attributes—lists, dictionaries, and even classes and instances. Here, the `Wrapper` class simply prints a trace message on each attribute access and delegates the attribute request to the embedded wrapped object:

```
>>> from trace import Wrapper
>>> x = Wrapper([1, 2, 3])          # Wrap a list
>>> x.append(4)                     # Delegate to list method
Trace: append
>>> x.wrapped                       # Print my member
[1, 2, 3, 4]

>>> x = Wrapper({'a': 1, 'b': 2})  # Wrap a dictionary
>>> list(x.keys())                  # Delegate to dictionary method
Trace: keys
['a', 'b']
```

The net effect is to augment the entire interface of the wrapped object with additional code in the `Wrapper` class. We can use this to log our method calls, route method calls to extra or custom logic, adapt a class to a new interface, and so on.

In the next chapter, we'll revive the notions of wrapped objects and delegated operations as one way to extend built-in types. If you are interested in the delegation design pattern, also watch for the discussions in [Chapters 32](#) and [39](#) of *function decorators*, a strongly related concept designed to augment a specific function or method call rather than the entire interface of an object, as well as *class decorators*, which serve as a way to automatically add such delegation-based wrappers to all instances of a class.



Delegation reminder: As noted in the sidebar “[Delegating Built-ins—or Not](#)” on [page 698](#), general proxies like the `Wrapper` example here cannot directly intercept and delegate calls to operator-overloading methods run by *built-in* operations. The list’s `__add__`, for instance, is not caught and fails:

```
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
>>> [1, 2, 3].__add__([4, 5])
[1, 2, 3, 4, 5]
>>> Wrapper([1, 2, 3]) + [4, 5]
TypeError: unsupported operand type(s) for +: 'Wrapper' and 'list'
```

Explicit-name attribute fetches are routed to `__getattr__`, but built-in operations differ in ways that impact some delegation-based tools. We’ll return to this issue and see it live in [Chapters 38](#) and [39](#), in the context of managed attributes and decorators. For now, keep in mind that delegation proxies may need to redefine operator-overloading methods by code, tools, or superclasses if those methods are used by embedded objects and should be routed to them.

Pseudoprivate Class Attributes

Besides larger structuring goals, class designs often must address name usage too. In [Chapter 28](#)’s case study, for example, we noted that methods defined within a general tool class might be modified by subclasses if exposed, and noted the trade-offs of this policy—while it supports method customization and direct calls, it’s also open to accidental replacements.

In [Part V](#), we learned that every name assigned at the top level of a module file is exported. By default, the same holds for classes—*data hiding* is a convention, and clients may fetch or change attributes in any class or instance to which they have a reference. All attributes are all “public” and “virtual,” in C++ terms: they’re accessible everywhere and are looked up dynamically at runtime. In fact, it’s even possible to change or delete a class’s method at runtime, though this is rarely done in practical programs. As a scripting language, Python is about enabling, not restricting.

All that being said, Python today does support the notion of name “mangling” (i.e., expansion) to associate some names with their classes. Mangled names are sometimes misleadingly called “private attributes,” but really this is just a way to *localize* a name to the class that created it—name mangling does not prevent access by code outside the class. This feature is mostly intended to avoid namespace *collisions* in instances, not to restrict access to names in general; mangled names are therefore better called “pseudoprivate” than “private.”

Pseudoprivate names are an advanced and entirely optional feature, and you probably won’t find them very useful until you start writing general tools or larger class hierarchies for use in multiprogrammer projects. In fact, they are not always used even when they probably *should* be—more commonly, Python programmers code internal names with a single underscore (e.g., `_X`), which is just an informal convention to let you know that a name shouldn’t generally be changed, but this means nothing to Python itself.

Because you may see this feature in other people’s code, though, you need to be somewhat aware of it, even if you don’t use it yourself. And once you learn its advantages and contexts of use, you may find this feature to be more useful in your own code than some programmers realize.

Name Mangling Overview

Here's how name mangling works: within a class statement only, any names that *start* with two underscores but do not *end* with two underscores are automatically expanded to include the name of the enclosing class at their front. For instance, a name like `__X` within a class named `Hack` is changed to `_Hack__X` automatically: the original name is prefixed with a single underscore and the enclosing class's name. Because the modified name contains the name of the enclosing class, it's generally unique; it won't clash with similar names created by other classes in a hierarchy.

Name mangling happens only for names that appear inside a class statement's code and then only for names that begin with two leading underscores. It works for *every* name preceded with double underscores, though—both class attributes (including method names) and instance attribute names assigned to `self`. For example, in a class named `Hack`, a method named `__meth` is mangled to `_Hack__meth`, and an instance attribute reference `self.__X` is transformed to `self._Hack__X`.

Despite the mangling, as long as the class uses the double-underscore version everywhere it refers to the name, all its references will still work. Because more than one class may add attributes to an instance, though, this mangling helps avoid clashes—but we need to move on to an example to see how.

Why Use Pseudoprivate Attributes?

One of the main issues that the pseudoprivate attribute feature is meant to alleviate has to do with the way instance attributes are stored. In Python, instance attributes normally wind up in the *single* instance object at the bottom of the class tree and are shared by class-level method functions the instance is passed into. This is different from the C++ model, where each class gets its own space for data members it defines.

Within a class's method in Python, whenever a method assigns to a `self` attribute (e.g., `self.attr = value`), it changes or creates an attribute in the instance (recall that inheritance searches happen only on reference, not on assignment). Because this is true even if multiple classes in a hierarchy assign to the same attribute, collisions are possible.

For example, suppose that when a programmer codes a class, it is assumed that the class owns the attribute name `X` in the instance. In this class's methods, the name is set and later fetched:

```
class C1:
    def meth1(self): self.X = 88          # I assume X is mine
    def meth2(self): print(self.X)
```

Suppose further that another programmer, working in isolation, makes the same assumption in another class:

```
class C2:
    def metha(self): self.X = 99          # Me too
    def methb(self): print(self.X)
```

Both of these classes work by themselves. The problem arises if the two classes are ever mixed together in the same class tree, using the multiple inheritance we'll expand on ahead:

```
class C3(C1, C2): ...
I = C3()                                # But only 1 X in I!
```

Now, the value that each class gets back when it says `self.X` will depend on which class assigned it *last*. Because all assignments to `self.X` refer to the same single instance, there is only one `X` attribute—I.X—no matter how many classes use that attribute name.

This isn't a problem if it's expected, and indeed, this is how classes normally *communicate*—the instance is shared memory. To guarantee that an attribute belongs to the class that uses it, though, prefix the name with double underscores everywhere it is used in the class, as in [Example 31-6](#), *pseudoprivate.py*.

Example 31-6. *pseudoprivate.py*

```
class C1:
    def meth1(self): self.__X = 88      # Now X is mine
    def meth2(self): print(self.__X)    # Becomes _C1__X in I
class C2:
    def metha(self): self.__X = 99      # Me too
    def methb(self): print(self.__X)    # Becomes _C2__X in I

class C3(C1, C2): pass
I = C3()                                # Two X names in I

I.meth1(); I.metha()                    # Set names
print(I.__dict__)                      # Actual storage
I.meth2(); I.methb()                   # Fetch names
```

When thus prefixed, the X attributes will be expanded to include the names of their classes before being added to the instance. If you run a `dir` call on I or inspect its namespace dictionary after the attributes have been assigned, you'll see the expanded names, `_C1__X` and `_C2__X`, but not X. Because the expansion makes the names more unique within the instance, the classes' coders can be fairly safe in assuming that they truly own any names that they prefix with two underscores:

```
$ python3 pseudoprivate.py
{'_C1__X': 88, '_C2__X': 99}
88
99
```

This trick can avoid potential name collisions in the instance, but note that it does not amount to true privacy. If you know the name of the enclosing class, you can still access either of these attributes anywhere you have a reference to the instance by using the fully expanded name (e.g., `I._C1__X = 77`). Moreover, names could still collide if unknowing programmers use the expanded naming pattern explicitly (unlikely, but not impossible). On the other hand, this feature makes it much less likely that you will *accidentally* step on a class's names.

Pseudoprivate attributes are also useful in larger frameworks or tools, both to avoid introducing new method names that might accidentally hide definitions elsewhere in the class tree and to reduce the chance of internal methods being replaced by names defined lower in the tree. If a method is intended for use only within a class that may be mixed into other classes, the double underscore prefix virtually ensures that the method won't interfere with other names in the tree, especially in multiple-inheritance scenarios:

```
class Super:
    def method(self): ...                # A real application method

class Tool:
    def __method(self): ...              # Becomes _Tool__method
    def other(self): self.__method()    # Use my internal method

class Sub1(Tool, Super): ...
    def actions(self): self.method()    # Runs Super.method as expected

class Sub2(Tool):
```



```
def __init__(self): self.method = 99    # Doesn't break Tool.__method
def method(self): ...                  # Ditto
```

We met multiple inheritance briefly in [Chapter 26](#) and will explore it in more detail later in this chapter. Recall that superclasses are searched according to their left-to-right order in class header lines. Here, this means Sub1 prefers Tool attributes to those in Super. Although in this example we could force Python to pick the application class's methods first by switching the order of the superclasses listed in the Sub1 class header, pseudoprivate attributes resolve the issue altogether. Pseudoprivate names also prevent subclasses from accidentally redefining the internal method's names, as in Sub2.

Again, this feature tends to be of use primarily for larger multiprogrammer projects and then only for selected names. Don't be tempted to clutter your code unnecessarily; only use this feature for names that truly need to be controlled by a single class. Although useful in some general class-based tools, for simpler programs, it's probably overkill.

For more examples that make use of the `__X` naming feature, see the *lister.py* mix-in classes introduced later in this chapter's multiple inheritance section, as well as the later class decorators mentioned in the following note.



Private matters: If you're interested in more binding forms of privacy, you may want to review the emulation of private instance attributes coded in “[Attribute Access: `\_\_getattr\_\_` and `\_\_setattr\_\_`](#)” on page 759 in [Chapter 30](#) and watch for the broader Private class decorator we'll build with delegation in [Chapter 39](#). Although it's possible to add name-access controls in Python classes, this is rarely done in practice—even for large systems that solve real-world problems. Go figure?

Method Objects: Bound or Not

Methods in general, and bound methods in particular, simplify the implementation of many design goals in Python. We met bound methods briefly while studying `__call__` in [Chapter 30](#). The full story, which we'll flesh out here, turns out to be more general and flexible than you might expect.

In [Chapter 19](#), we learned how functions can be processed as normal objects. Methods are a kind of object too, and can be used generically in much the same way as other objects—they can be assigned to names, passed to functions, stored in data structures, and so on—and like simple functions, modules, and classes, qualify as *first-class objects*. Because a class's methods can be accessed from an instance or a class, though, they come in two flavors:

Bound methods: when a method is referenced through an instance

Accessing a function attribute of a class by qualifying an *instance* returns a bound method object. This object automatically packages the instance with the function as a pair. When a bound method is later called, the instance is automatically passed to the function's `self` argument.

Plain functions: when a method is referenced through a class

Accessing a function attribute of a class by qualifying a *class* returns a plain function object. To call this function later, you must provide an instance object explicitly to the `self` argument—if the function expects one.

Both kinds of methods are full-fledged objects; they can be transferred around a program at will, just like strings and numbers. Bound methods simply remember an instance, but plain functions do not. This is why we've had to pass in an instance explicitly when calling superclass methods from subclass

methods in previous examples (including this chapter's *employees.py* in [Example 31-1](#)); technically, such calls produce plain functions along the way.

When calling a *bound* method object, though, Python provides the instance argument for us—the instance that was used to create the bound method object. This means that bound method objects are usually interchangeable with simple function objects and makes them especially useful for interfaces originally written for functions.



Blast from the past: Python once required all methods in a class to have an instance argument and termed methods fetched directly from a class *unbound methods*—implying that they required an instance argument when later called. Today, method functions in a class are really just plain functions; their only special quality is that they are bound to an instance when fetched through the instance. Unbound methods, however, are dead; long live plain functions!

Bound Methods in Action

To illustrate in simple terms, suppose we define the following class in the REPL of our choosing:

```
>>> class Hack:
    def doit(self, message):
        print(message)
```

Now, in normal operation, we make an instance and call its method in a single step to print the passed-in argument:

```
>>> inst = Hack()
>>> inst.doit('hello')    # Typical method calls
hello
```

Really, though, a *bound* method object is generated along the way—just before the method call's parentheses. In fact, we can fetch a bound method without actually calling it. An *object.name* expression evaluates to an object as all expressions do. In the following, it returns a bound method object that packages the instance (*inst*) with the method function (*Hack.doit*). The net effect is that we can assign this bound method pair to another name and then call it as though it were a simple self-less function:

```
>>> inst = Hack()
>>> meth = inst.doit      # Bound method object: instance+function
>>> meth('hola')         # Same effect as inst.doit('...')
hola
```

In fact, if you know where to look, you can see the instance/function pair that the bound method packages:

```
>>> meth
<bound method Hack.doit of <__main__.Hack object at 0x108f2e8a0>>
>>> meth.__self__
<__main__.Hack object at 0x108f2e8a0>
>>> meth.__func__
<function Hack.doit at 0x108f37ce0>
```

On the other hand, if we qualify the *class* to get to *doit*, we get back a plain function object with no associated instance. To call this type of method, we usually must pass in an instance as the leftmost argument—there isn't one in the expression otherwise, and the method in this demo expects it:

```
>>> inst = Hack()
>>> meth = Hack.doit      # Plain function: requires self
>>> meth(inst, 'ciao')    # Pass in instance (if the method expects one)
ciao
```

This time, though, the method is substantially more mundane because it's a plain function:

```
>>> meth
<function Hack.doit at 0x108f37ce0>
```

By extension, the same rules apply within a class's method if we reference `self` attributes that name functions in a class. A `self.method` expression is a bound method object because `self` is an instance object, though a `class.method` is a simple function that may require a `self` when run:

```
>>> class Hack2(Hack):
    def doit2(self):
        meth = self.doit      # Bound method object: instance+function
        meth('bonjour')      # Looks like a simple function
        meth = Hack.doit
        meth(self, 'privit')  # Plain function: requires self

>>> Hack2().doit2()
bonjour
privit
```

Most of the time, you call methods immediately after fetching them with attribute qualification, so you don't notice the method objects generated along the way. But if you start writing code that calls objects generically, you need to be careful to treat nonbound methods specially—they normally require an explicit instance object to be passed in.

Implied by this model, classes can also code methods that *do not* require a `self` instance, and are called normally when fetched from their *class*; apart from inheritance, this is similar to a function in a module file:

```
>>> class Hack3:
    def doit3(message):      # A self-less method (function)
        print(message)

>>> Hack3.doit3('guten tag')  # No "self" is passed
guten tag
```

This falls down, though, if we try to call such a self-less method function through an *instance*: because the bound method made along the way packages and passes an instance, the call sends one too many arguments:

```
>>> x = Hack3()
>>> x.doit3('namaste')
TypeError: Hack3.doit3() takes 1 positional argument but 2 were given
```

In other words, you can code self-less functions in a class, and can call them normally through a class without an instance. To call them through an instance, too, though, you'll have to stay tuned for the next chapter's coverage of *static* and *class* methods—methods marked specially to suppress an automatic instance argument in all contexts. Also in the next chapter, you'll see that the `super` built-in binds an instance with a method, too, but is substantially more convoluted than the bound method pairs we've met here.

Finally, to demo how flexible bound methods can be, the following stores four of them in a list and calls them generically, with normal call expressions:

```

>>> class Number:
    def __init__(self, base):
        self.base = base
    def double(self):
        return self.base * 2
    def triple(self):
        return self.base * 3

>>> x, y, z = Number(2), Number(3), Number(4)           # Class instance objects
>>> x.double()                                           # Normal immediate calls
4
>>> acts = [x.double, y.double, z.double, z.triple]     # List of bound methods
>>> for act in acts:                                    # Calls are deferred
    print(act(), end=' ')                                # Call as though functions

4 6 8 12

```

In the end, calls to `act` here run methods in the class to process instances—both saved previously in bound methods.

Why You Will Care: Bound Method Callbacks

Because bound methods automatically pair an instance with a class's method function, you can use them anywhere a simple function is expected. One of the most common places you'll see this idea put to work is in code that registers methods to handle event callbacks run by GUI interfaces, like Python's `tkinter` standard-library module. As review, here's the simple case:

```

def handler():
    ...use globals or closure scopes for state...
...
widget = Button(text='Tap', command=handler)

```

With `tkinter`, to register a handler for button click events, we usually pass a callable object that takes no arguments to the `command` keyword argument. Function names (and `lambdas`) work here, and so do class-level methods—though they must be bound methods if they expect an instance when called:

```

class MyGui:
    def handler(self):
        ...use self.attr for state...
    def makewidgets(self):
        b = Button(text='Tap', command=self.handler)

```

Here, the event handler is `self.handler`—a bound method object that remembers both `self` and `MyGui.handler`. Because `self` will refer to the original instance when `handler` is later invoked on events, the method will have access to instance attributes that can retain state between events, as well as class-level methods. With simple functions, state normally must be retained in global variables or enclosing function scopes (a.k.a. closures) instead.

See also the discussion of `__call__` operator overloading in [Chapter 30](#) for another way to make classes compatible with function-based APIs, as well as `lambda` in [Chapter 19](#) for another tool often used in callback roles. As noted in the former of these, you don't need to wrap a bound method in a `lambda` unless extra arguments must be added to the call; because the bound method in the preceding example *already* defers the call (there are no parentheses to trigger one!), adding a `lambda` here would otherwise be pointless.

Classes Are Objects: Generic Object Factories

Sometimes, class-based designs require objects to be created in response to conditions that can't be predicted when a program is written. The factory design pattern allows such a deferred approach. Due in large part to Python's flexibility, factories can take multiple forms, some of which don't seem special at all.

Because classes are also first-class objects (in the [Chapter 19](#) sense), it's easy to pass them around a program, store them in data structures, and so on. You can also pass classes to functions that generate arbitrary kinds of objects; such functions are sometimes called *factories* in OOP design circles. Factories can be a major undertaking in a statically typed language such as C++ but are almost trivial to implement in Python.

For example, the call syntax we studied in [Chapter 18](#) can call any class with any number of positional or keyword constructor arguments in one step to generate any sort of instance. [Example 31-7](#) demos the underlying code.

Example 31-7. factory.py

```
def factory(aClass, *pargs, **kargs):           # Varargs tuple, dict
    return aClass(*pargs, **kargs)             # Call aClass

class Hack:
    def doit(self, message):
        print(message)

class Person:
    def __init__(self, name, job=None):
        self.name = name
        self.job = job

object1 = factory(Hack)                        # Make a Hack object
object2 = factory(Person, 'Sue', 'dev')        # Make a Person object
object3 = factory(Person, name='Bob')          # Ditto, with keywords and default
```

This code's factory is passed a class object, along with zero or more arguments for the class's constructor. When called, it uses star syntax to collect and unpack arguments and calls the class to return an instance. Really, `factory` can invoke any callable object, including functions, classes, and methods, but we're using it for classes here.

The rest of the example simply defines two classes and generates instances of both by passing them to the `factory` function. And that's the only factory function you may ever need to write in Python; it works for any class and any constructor arguments. If you run this example live, your objects will look like this:

```
>>> from factory import *
>>> object1.doit(99)
99
>>> object2.name, object2.job
('Sue', 'dev')
>>> object3.name, object3.job
('Bob', None)
```

By now, you should know that everything is a first-class object in Python—including classes, which are usually just compiler input in languages like C++. It's natural to pass them around this way. As

mentioned at the start of this part of the book, though, only objects *derived* from classes do full OOP in Python.

Why Factories?

So, what good is the factory function (besides providing an excuse to illustrate first-class class objects in this book)? Unfortunately, it's difficult to show applications of this design pattern without listing much more code than we have space for here. In general, though, such a factory might allow code to be insulated from the details of dynamically configured object construction.

For instance, recall the `Processor` class presented as a composition demo earlier in [Example 31-3](#). It accepts reader and writer objects for processing arbitrary data streams. The original abstract version of this example in [Chapter 26](#) manually passed in instances of specialized classes like `FileWriter` and `SocketReader` to customize the data streams being processed; later, we passed in hardcoded file, stream, and formatter objects. In a more dynamic scenario, external devices such as configuration files or GUIs might be used to configure the streams.

In such a dynamic world, we might not be able to hardcode the creation of stream interface objects in our scripts but might instead create them at runtime according to the contents of a configuration file.

Such a file might simply give the string name of a stream class to be imported from a module, plus an optional constructor call argument. Factory-style functions or code might come in handy here because they would allow us to fetch and pass in classes that are not hardcoded in our program ahead of time. Indeed, those classes might not even have existed at all when we wrote our code. Hypothetically:

```
classname = ...parse from config file...
classarg   = ...parse from config file...

import streamtypes                # Customizable code
aclass = getattr(streamtypes, classname) # Fetch from module
reader = factory(aclass, classarg)      # Or aclass(classarg)
processor(reader, ...)
```

Here, the `getattr` built-in is again used to fetch a module attribute given a string name (it's like saying `obj.attr`, but `attr` is a string). This code snippet doesn't strictly need `factory`—it could make an instance with just `aclass(classarg)`. A separate function, though, may prove more useful when extra work is required at instance creation time, such as caching objects for reuse. However they are coded, factories are almost trivial with Python's dynamic typing and universal first-class object model.

Multiple Inheritance and the MRO

Our last design pattern is one of the most useful and will serve as a subject for more realistic examples to wrap up this chapter. As a bonus, the code we'll write here may be useful tools.

Most of our examples so far have used *single inheritance*—class trees in which each class has just one superclass. This suffices for simple hierarchies and enables customization. In our pizza shop demo of [Example 31-1](#), for example, each worker belonged to just one category and inherited names from only one branch of the class tree. Abstractly:

```
class B: ...
class A(B): ...
I = A()                # Use attributes from I, A, and B - in that order
```

Many class-based designs, however, call for *combining* disparate sets of methods. As we've seen, in a `class` statement, more than one superclass can be listed in parentheses in the header line. When you do

this, you leverage *multiple inheritance*—the class and its instances inherit names from *all* the listed superclasses:

```
class C: ...
class B: ...
class A(B, C): ...
I = A()                # Use attributes from I, A, B, and C - in that order
```

In general, multiple inheritance is good for modeling objects that belong to more than one set. For instance, a person may be an engineer, a writer, a musician, and so on and inherit properties from all such sets. With multiple inheritance, objects obtain the *union* of the behavior in all their superclasses. As you'll see ahead, multiple inheritance also allows classes to function as general packages of mixable attributes known as *mix-in* classes.

Although it's a useful tool, multiple inheritance adds another dimension to attribute inheritance:

- *Single* inheritance searches only *depth first*—from instance, to class, and to each superclass from lowest to highest. This order is determined by the class from which an instance is made and the superclass listed in parentheses in each class statement's header (which is mirrored by class objects' `__bases__` attributes).
- *Multiple* inheritance also searches *left to right*—according to the order of classes listed in parentheses in class headers. This is a nested component, pursued only after branches further to the left have reached the top of the tree and have been exhausted of their own right-branch candidates.

We call the combination of these two *DFLR*, for depth first, and then left to right. This search suffices when an attribute name shows up in just one branch of a class tree, which is a typical case. When the same name appears in multiple branches, though, DFLR alone is subpar: a lower (and hence more specialized) subclass to the right isn't able to redefine a name in one of its higher (and hence more general) superclasses reached through a branch to the left.

Because of this, multiple inheritance requires a slightly different search order known as *MRO*, for method resolution order (though it's used for all attributes, not just methods). In brief, MRO order works the same as DFLR in typical trees but proceeds *across* by tree levels before moving up in a more breadth-first fashion when multiple classes in a tree share a common superclass, forming what's called a *diamond* pattern—after the tree's square-on-its-corner shape.

This MRO search order is run for all class trees but differs from DFLR when multiple-inheritance diamonds are present. It's designed to visit a common superclass just *once*, and *after* all its subclasses. While user-defined classes don't often form diamonds in Python, the built-in object class automatically added above all *root* (topmost) classes makes every multiple-inheritance tree a diamond; without MRO, object's defaults may hide user-defined versions.

To illustrate what diamonds are all about and how MRO search runs across before up in this one somewhat atypical case, let's turn to some code.

How Multiple Inheritance Works

We'll get more formal about MROs in the next section, but it's easy to demo live. Let's start with the simple and typical case. In the following, class A inherits from both B and C using multiple inheritance, and B and C are root (topmost) classes that define unique attributes. When we make an instance I of A and ask for its attributes, inheritance searches I, A, B, and C—and in that order. Hence, `attr1` is located in B, and `attr2` in C (if you're working along in a REPL, type a blank line after each `class`, omitted here for brevity):

```
>>> class C:      attr2 = 'C2'
>>> class B:      attr1 = 'B1'
>>> class A(B, C): pass

>>> I = A()
>>> I.attr1, I.attr2
('B1', 'C2')
```

As you might already expect, B wins if it has the *same* name as C due to the left-to-right component of the search, and lower classes like A still beat their supers as before:

```
>>> class C:      attr2 = 'C2'
>>> class B:      attr2 = 'B2'; attr1 = 'B1'
>>> class A(B, C): attr1 = 'A1'

>>> I = A()
>>> I.attr1, I.attr2
('A1', 'B2')
```

Next, let's add a higher superclass above B: in the following, A inherits from both B and C as before, but B also inherits from D, and C is a root class with no supers. Per the DFLR order (depth first and then left to right), attr2 is found in D by virtue of searching I, A, B, and D—and before C is even checked:

```
>>> class D:      attr2 = 'D2'
>>> class C:      attr2 = 'C2'
>>> class B(D):   attr1 = 'B1'
>>> class A(B, C): pass

>>> I = A()
>>> I.attr1, I.attr2
('B1', 'D2')
```

Watch what happens, though, if B and C *both* inherit from D, and C has the same attribute as D—per the MRO, the lower C class's attribute wins and effectively replaces that attribute in D:

```
>>> class D:      attr2 = 'D2'
>>> class C(D):   attr2 = 'C2'
>>> class B(D):   attr1 = 'B1'
>>> class A(B, C): pass

>>> I = A()
>>> I.attr1, I.attr2
('B1', 'C2')
```

This last case is what is meant by a *diamond* pattern of inheritance: multiple classes, B and C, both inherit from the same superclass. When this happens, the lower class's version of an attribute is used instead of a same-named version in a superclass—even if that superclass could be reached first by depth alone.

Importantly, this differs only in diamonds: *nondiamonds* still choose higher superclasses over lower classes with the same attribute name as in the third example in this section. The MRO allows a class to override attributes in a higher class from which it inherits; in the last example, C can replace names in D even if D is first per DFLR.

To summarize: in *nondiamonds*, the search proceeds all the way to the top, then backs up and starts searching to the right (DFLR); in *diamonds*, the search checks classes to the right before climbing up to a common superclass (MRO).

Strictly speaking, all four examples in this section are *implicit* diamonds because root classes inherit from the built-in object class automatically. We’ve seen that this class provides defaults for some methods, such as print displays. Because of the MRO, the object built-in’s defaults never hide methods in user-defined classes below it in a class tree. To see how object is ruled out this way, let’s get a bit more precise on the MRO’s order.

How the MRO Works

Formally speaking, the MRO inheritance search order works as if all classes are listed per the DFLR, and then all but the rightmost duplicate of each class is removed. In more detail, it’s computed as follows:

1. List all the classes from which an instance inherits using the DFLR lookup order, and include a class multiple times if it’s visited more than once.
2. Scan the resulting list for duplicate classes, removing all but the last (rightmost) occurrence of duplicates in the list.

The resulting MRO sequence for a given class includes the class, its superclasses, and all higher superclasses up to and including the implicit or explicit object root class above the tops of the tree. It’s ordered such that each class appears before its parents, and multiple parents retain the order in which they appear in the class header.

Because common parents in diamonds appear only at the position of their *last* visitation in the MRO, lower classes are searched first when the MRO list is used later by attribute inheritance (making it more breadth-first than depth-first in diamonds only), and each class is included and thus visited just once, no matter how many classes lead to it.

Consider the *nondiamond* class tree of [Example 31-8](#), whose shape is sketched as “ASCII art” in its comments.

Example 31-8. mro_nondiamond.py

```
class E:      attr = 'E'      # D      E
class D:      attr = 'D'      # |      |
class C(E):   attr = 'C'      # B      C
class B(D):   pass           # \      /
class A(B, C): pass          #   A
                        #   |
X = A()       #           X
print(X.attr) # D
```

To compute the MRO inheritance search order through this tree, Python first enumerates all classes accessible per DFLR and then removes duplicates (remember that the built-in object is implicitly above all root classes):

```
DFLR => [X, A, B, D, object, C, E, object]
MRO  => [X, A, B, D, C, E, object]
```

In this tree, the net result for both DFLR and MRO ordering is the same: the output is “D” by taking attr from class D at the top left. Technically, the MRO differs because the built-in object appears just once at the end after its duplicates are removed; this is irrelevant to our classes because they don’t redefine an object default (like print strings). When a user-defined *diamond* is coded in [Example 31-9](#), however, the MRO’s difference is more striking.

Example 31-9. *mro_diamond.py*

```
class D:      attr = 'D'      #      D
class C(D):   attr = 'C'      #    /  \
class B(D):   pass           #   B    C
class A(B, C): pass         #    \  /
               #      A
X = A()      #      |
print(X.attr) # C          #      X
```

For this tree, the common user-defined class D is reached *twice* by DFLR. Because the MRO keeps only the last (rightmost) D, the lower C’s definition of `attr` now wins over D’s. Hence, the output is now “C” instead of “D”:

```
DFLR => [X, A, B, D, object, C, D, object]
MRO  => [X, A, B, C, D, object]
```

In fact, you can view the MRO of any class with its built-in `__mro__` attribute. This tuple gives the inheritance search order followed for instances of the class (after the instance itself). It’s set to the result of the `mro` class method at class creation time (which can technically be customized for roles too obscure to cover here). It’s also a lot to look at unless we select class names as in the following, which inspects classes in the *nondiamond* tree of [Example 31-8](#):

```
>>> from mro_nondiamond import *
D
>>> [c.__name__ for c in A.__mro__]
['A', 'B', 'D', 'C', 'E', 'object']
>>> [c.__name__ for c in C.__mro__]
['C', 'E', 'object']
```

And here’s the case for the *diamond* class tree in [Example 31-9](#):

```
>>> from mro_diamond import *
C
>>> [c.__name__ for c in A.__mro__]
['A', 'B', 'C', 'D', 'object']
>>> [c.__name__ for c in C.__mro__]
['C', 'D', 'object']
```

The full `__mro__` is the classes used by inheritance at a given class, and `__bases__` is just supers there:

```
>>> A.__mro__
(<class 'mro_diamond.A'>, <class 'mro_diamond.B'>, <class 'mro_diamond.C'>,
 <class 'mro_diamond.D'>, <class 'object'>)

>>> X.__class__.__mro__
(<class 'mro_diamond.A'>, <class 'mro_diamond.B'>, <class 'mro_diamond.C'>,
 <class 'mro_diamond.D'>, <class 'object'>)

>>> A.__bases__
(<class 'mro_diamond.B'>, <class 'mro_diamond.C'>)

>>> D.__bases__
(<class 'object'>,)

```

Experiment with `__mro__` on your own for more fidelity. Especially if you’re unsure how the MRO handles a given class tree, this attribute can be consulted to see how inheritance will truly search.

Attribute Conflict Resolution

Though a useful pattern, multiple inheritance’s chief downside is that it can pose a *conflict* when the same method (or other attribute) name is defined in more than one branch of the class tree. When this occurs, the conflict is resolved either automatically by the inheritance search order or manually in your code:

Default

By default, inheritance chooses the *first* occurrence of an attribute it finds when an attribute is referenced normally (e.g., by `self.attr`). In this mode, Python chooses the first appearance located while scanning the MRO of an instance’s class, from left to right.

Explicit

In some class models, you may need to *select* an attribute explicitly by referencing it through its class name (e.g., by `superclass.attr`). Your code resolves the conflict this way and overrides the search’s default to select an option other than the inheritance search’s default.

By default, multiple inheritance assumes that names on the *left* of a class tree should override the same names on the right, and the MRO further assumes that *lower* classes should override same-named attributes in common superclasses of diamonds. Of course, the problem with assumptions is that they assume things. In [Example 31-9](#), what if you need some same-named attributes from B but *also* some others from C?

Luckily, there is an inheritance escape hatch. If the default search order doesn’t work, or if you simply want more control over the search process, you can always force the selection of an attribute from anywhere in the tree by assigning or otherwise naming the one you want at the place where classes are mixed together.

In [Example 31-9](#), for instance, any of the following selections are allowed, but the latter two make the choice explicit rather than relying on the implicit and subtle “magic” of MRO inheritance:

```
class A(B, C): pass           # Use the default MRO choice (C)
class A(B, C): attr = B.attr  # Choose attr from the left branch (D)
class A(B, C): attr = C.attr  # Choose attr from the right branch (C)
```

Naturally, attributes picked this way can also be methods—which are normal, assignable attributes that happen to reference callable function objects. Moreover, the choice can be made in a call:

```
class A(B, C):
    def ...:
        self.method(...)      # Use the default MRO choice

class A(B, C):
    def ...:
        B.method(self, ...)    # Choose method from the left branch

class A(B, C):
    def ...:
        C.method(self, ...)    # Choose method from the right branch
```

Such calls can be used to override the MRO, but also to kick calls up the tree when the class’s own version must be skipped. As you’ll see in the next chapter, such calls might also use `super().method()`, which selects the class following the call’s host on `self`’s MRO; though simple in single inheritance, this can be stunningly implicit in multiple inheritance and doesn’t provide as much control as explicit class names.

However they are coded, such manual overrides are required only when the *same name* appears in multiple superclasses, and you do not wish to use the first one inherited. For example, manual overrides allow you to unambiguously choose among a set of same names in *both* left and right branches, while inheritance would choose just names on the left. They can also be used to make choices explicit in general, though, even in nondiamonds.

Because this isn't as common an issue in typical Python code as it may sound, we'll defer details on this topic until we study the super built-in in the next chapter and revisit this as a "gotcha" at the end of that chapter. First, though, the next section demonstrates a practical use case for the multiple-inheritance design pattern.



The inheritance finale: Despite the MRO's complexity, there are still a few inheritance hurdles left to clear. Its complete algorithm is more complex than the model sketched here, incorporating special cases for metaclasses, descriptors, and built-ins. In [Chapter 32](#), we'll expand inheritance for the metaclass tree and apply its MRO in the super built-in, and in [Chapter 38](#), we'll study its special case for built-in operations, but we won't be able to formalize inheritance in full until [Chapter 40](#) after we've studied all these tools in more depth. The good news is that the remaining hurdles almost never trip up application programs; for most coders, the "finale" is a fully optional read.

Example: "Mix-in" Attribute Listers

Perhaps the most common way multiple inheritance is used is to "mix in" general-purpose methods from superclasses. Such superclasses are usually called *mix-in classes*—they provide methods you add to application classes by inheritance. In a sense, mix-in classes are similar to modules: they provide packages of methods for use in their client subclasses. Unlike simple functions in modules, though, methods in mix-in classes also can participate in inheritance hierarchies and have access to the self instance for using state information and other methods in their trees.

For example, we've seen that Python's default way to print a class instance object isn't incredibly useful:

```
>>> class Hack:
    def __init__(self, what):          # No __repr__ or __str__
        self.data1 = what

>>> X = Hack('code')
>>> print(X)                          # Default: class name + address (id)
<__main__.Hack object at 0x10ba464e0>
```

As you learned in both [Chapter 28](#)'s case study and [Chapter 30](#)'s operator-overloading coverage, classes can provide a `__str__` or `__repr__` method to implement custom displays. But rather than coding one of these in each class you wish to print, why not code it once in a general-purpose tool class and inherit it in all your other classes?

That's what mix-ins are for. Defining a display method in a mix-in superclass once enables us to reuse it anywhere we want to see a custom display format—even in classes that may already have another superclass. We've already explored tools that do related work:

- Chapter 28's `AttrDisplay` class, of [Example 28-13](#), formatted instance attributes in a generic `__repr__` method, but it did not climb class trees and was utilized in single-inheritance mode only.
- Chapter 29's `classtree.py` module, of [Example 29-9](#), defined functions for climbing and sketching class trees, but it did not display object attributes along the way and was not architected as an inheritable class.

Here, we're going to revisit these examples' techniques and expand upon them to code a set of three mix-in classes that serve as generic display tools for listing instance attributes, inherited attributes, and attributes on all objects in a class tree, respectively. We'll also use our tools in multiple-inheritance mode and deploy coding techniques that make classes better suited to use as generic tools.

Unlike [Chapter 28](#), we'll code this with a `__str__` instead of a `__repr__`. This is partially a style issue and limits their role to `print` and `str`, but the displays we'll be developing are meant to be user-friendly, not imitative of code. This policy also leaves client classes the option of coding an alternative lower-level display for interactive echoes and nested appearances with a `__repr__` and has built-in immunity from `__repr__` looping perils covered ahead.

Listing instance attributes with `__dict__`

Let's get started with the simple case—listing attributes attached to an instance. [Example 31-10](#), coded in the file `listinstance.py`, defines a mix-in called `ListInstance` that overloads the `__str__` method for all classes that include it in their header lines. Because this is coded as a class, `ListInstance` is a generic tool whose formatting logic can be used for instances of any subclass client.

Example 31-10. `listinstance.py`

```
class ListInstance:
    """
    Mix-in class that provides a formatted print() or str() of instances via
    inheritance of __str__ coded here. Displays instance attrs only; self is
    instance of lowest class; __X naming avoids clashing with client's attrs.
    Works for classes with slots: a __dict__ is ensured by lack of slots here.
    """
    def __attrnames(self):
        result = '\n'
        for attr in sorted(self.__dict__):
            result += f'\t{attr}={self.__dict__[attr]!r}\n'
        return result

    def __str__(self):
        return (f'<Instance of {self.__class__.__name__}, '
                f'address {id(self):#x}: '
                f'{self.__attrnames()}>')

if __name__ == '__main__':
    import testmixin
    testmixin.testmixin(ListInstance)  # Test class in this module
```

The `__attrnames` method here exhibits a classic comprehension pattern, and you might save some program real estate by implementing it more concisely with a *generator expression* triggered by a `''.join()` call; we'll leave this as a suggested exercise. As coded, `ListInstance` uses some previously explored techniques to extract the instance's class name and attributes:

- It uses a `self.__class__.__name__` expression to fetch the name of an instance's class. Recall that each instance has a built-in `__class__` attribute that references the class from which it was created, and each class has a `__name__` attribute that references the class's name given in its header line.
- It does most of its work by simply scanning the instance's attribute dictionary (remember, it's available in `__dict__`) to build up a string showing the names and values of all instance attributes. The dictionary's keys are sorted by name to be human-friendly, instead of relying on the dictionary's insertion order.

In these respects, `ListInstance` is similar to [Chapter 28](#)'s attribute display; in fact, it's largely just a variation on a theme. Our class here, though, uses two additional techniques:

- It displays the instance's memory address by calling the `id` built-in function, which returns any object's address. By definition, this is a unique object identifier, which will be useful in later mutations of this code.
- It uses the *pseudoprivate* naming pattern for its worker method: `__attrnames`. As we saw earlier in this chapter, Python automatically localizes any such name to its enclosing class by expanding the attribute name to include the class name; in this case, it becomes `_ListInstance__attrnames`. This holds true for both class attributes like methods and instance attributes attached to `self`. As first noted in [Chapter 28](#), this helps in a general tool like this, as it ensures that its names won't clash with any names used in its client subclasses.

Because `ListInstance` defines a `__str__` operator-overloading method, instances derived from this class display their attributes automatically when printed, giving a bit more information than a simple address. Here is this tool class in action in single-inheritance mode, mixed in to the previous section's class:

```
>>> from listinstance import ListInstance
>>> class Hack(ListInstance):                    # Inherit a __str__ method
    def __init__(self, what):
        self.data1 = what

>>> X = Hack('code')
>>> print(X)                                     # print() and str() run __str__
<Instance of Hack, address 0x10c890b90:
    data1='code'
>
```

You can also get the listing display as a string with `str` without printing it (use `print` later to apply escapes), and interactive echoes and nesting still use the default format (we've left `__repr__` as an option for clients):

```
>>> str(X)
"<Instance of Hack, address 0x10c890b90:\n\tdata1='code'\n>"
>>> X
<__main__.Hack object at 0x10c890b90>
```

The `ListInstance` class is useful for any classes you write—even classes that already have one or more superclasses. This is where *multiple inheritance* comes in handy: by adding `ListInstance` to the list of superclasses in a class header (i.e., mixing it in), you get its `__str__` “for free” while still inheriting from the existing superclass(es). The file *testmixin.py* in [Example 31-11](#) codes test classes that prove the point.

Example 31-11. testmixin.py

```
"""
Generic lister-mixin tester: similar to transitive reloader in
Chapter 25, but passes a class object to tester (not function),
and testByNames adds loading of both module and class by name
strings here, in keeping with Chapter 31's factories pattern.
"""

import importlib

def tester(listerclass, sept=False):
    "Pass any lister class to listerclass"

    class Super:
        def __init__(self):
            self.data1 = 'code'
        def method1(self):
            pass

    class Sub(Super, listerclass):
        def __init__(self):
            Super.__init__(self)
            self.data2 = 'Python'
            self.data3 = 3.12
        def method2(self):
            pass

    instance = Sub()
    print(instance)
    if sept: print(f'\n{ '-' * 80 }\n')

def testByNames(modname, classname, sept=False):
    modobject = importlib.import_module(modname)
    listerclass = getattr(modobject, classname)
    tester(listerclass, sept)

if __name__ == '__main__':
    testByNames('listinstance', 'ListInstance', True)
    testByNames('listinherited', 'ListInherited', True)
    testByNames('listtree', 'ListTree', False)
```

This file is instrumented to allow us to test a variety of listers. To this end, it is passed a lister class, `listerclass`, to be mixed in with test classes nested in a function. Again, classes are passable first-class objects, and we need to make the lister class a parameter to allow it to vary per test. This file also has tools to fetch classes by name strings, which we'll set aside for the moment.

More important here is the self-test code in [Example 31-10](#): it imports the tester function in [Example 31-11](#) and passes in `ListInstance` to be tested here. Hence, `Sub` here inherits names from both `Super` and `ListInstance`; it's a *composite* of its own names and names in both its superclasses. When you make a `Sub` instance and print it, you automatically get the custom representation mixed in from `ListInstance`:

```
$ python3 listinstance.py
<Instance of Sub, address 0x10caae300:
  data1='code'
  data2='Python'
  data3=3.12
>
```

The `ListInstance` class responsible for this display works in any class it's mixed into because `self` refers to an instance of the subclass that pulls this class in, whatever that may be. Again, in a sense, mix-in classes are the class equivalent of modules—packages of methods useful in a variety of clients.

Besides the utility they provide, mix-ins optimize code maintenance, as all classes do. For example, if you later decide to extend `ListInstance`'s `__str__` to also print all the class attributes that an instance inherits, you're covered; because it's an inherited method, changing `__str__` automatically updates the display of each subclass that imports the class and mixes it in. And since it's now officially “later,” let's move on to the next section to see what such an extension might look like.

Listing inherited attributes with `dir`

As it is, our `ListInstance` mix-in displays instance attributes only—names attached to the instance object itself. It's nearly trivial, though, to extend the class to display *all* the attributes accessible from an instance—both its own and those it inherits from its classes. The trick is to use the `dir` built-in function instead of scanning the instance's `__dict__` dictionary; the latter holds instance attributes only, but the former also collects all inherited attributes.

The mutation in [Example 31-12](#), *listinherited.py*, codes this scheme. This is located in its own module to facilitate testing, but if existing clients were to use this version instead, they would pick up the new display automatically (and recall from [Chapter 25](#) that a `from import's` `as` clause can rename a new version to a prior name being used).

Example 31-12. *listinherited.py*

```
class ListInherited:
    """
    Use dir() to collect both instance attrs and names inherited from
    its classes. This includes default names inherited from the implied
    'object' superclass above topmost classes. getattr() can fetch
    inherited names not in self.__dict__.

    Caution: use __str__, not __repr__, or else this loops when printing
    bound methods that may be returned for some attributes by getattr().
    This will normally fail for class "slots" attributes not yet assigned.
    """

    def __attrnames(self, unders=False):
        result = '\n'
        for attr in dir(self):
            if attr[:2] == '__' and attr[-2:] == '__':
                result += f'\t{attr}\n' if unders else ' '
            else:
                result += f'\t{attr}={getattr(self, attr)!r}\n'
        return result

    def __str__(self):
        return (f'<Instance of {self.__class__.__name__}, '
                f'address {id(self):#x}: '
                f'{self.__attrnames()}>')

if __name__ == '__main__':
    import testmixin
    testmixin.testers(ListInherited)    # Test class in this module
```


Notice that this code, by default, skips all `__X__` names for brevity. There are 27 such names in the test case, and most of these are internal names that we don't generally care about in a generic listing like this. Some are user-defined operator-overloading methods like our `__str__`, though most reflect defaults in the implicit object root class, and there's no easy way to determine an attribute's class of origin here (but stay tuned).

Because `dir` results may include names from anywhere in a class tree, this version also must use the `getattr` built-in function to fetch attributes by name string instead of indexing the instance's `__dict__` attribute dictionary. `getattr` runs inheritance search, and some of the names we're listing here are not stored on the instance itself.

To test the new version, run its file directly—it passes the `ListInherited` class it defines to the *testmixin.py* file's test function in [Example 31-11](#) to be mixed in with a subclass in the function. Here's the output of this test and lister class; notice the extra names inherited from classes and the name mangling at work in the lister's method name:

```
$ python3 listinherited.py
<Instance of Sub, address 0x101d76600:
  _ListInherited__attrnames=<bound method ListInherited.__attrnames of <...>
  data1='code'
  data2='Python'
  data3=3.12
  method1=<bound method tester.<locals>.Super.method1 of <testmixin.tester...>
  method2=<bound method tester.<locals>.Sub.method2 of <testmixin.tester...>
>
```

The display of bound methods in this was truncated to fit this page; here's what the first two look like with an added line break (as usual, run on your own for the full and up-to-date picture):

```
_ListInherited__attrnames=<bound method ListInherited.__attrnames of
  <testmixin.tester.<locals>.Sub object at 0x101d76600>

method1=<bound method tester.<locals>.Super.method1 of
  <testmixin.tester.<locals>.Sub object at 0x101d76600>
```

Display formatting is an open-ended task (e.g., Python's standard `pprint` “pretty printer” module may offer options here too), so we'll leave further polishing as a suggested exercise. The tree lister of the next section may be more useful in any event, so let's move on.



Looping in `__repr__`: Now that we're displaying inherited methods too, we must use `__str__` instead of `__repr__` to overload printing. With `__repr__`, this code (and code like it) will fall into *recursive loops*—its `getattr` returns a bound method; whose display includes the instance; which triggers `__repr__` again to display the instance; and so on, quickly triggering a stack-overflow exception. Subtle, but true! Change `__str__` to `__repr__` to see this live. One way to avoid such `__repr__` loops is to skip `getattr` results for which `isinstance` comparisons to the standard library's `types.MethodType` are true. Using `__str__` instead is simpler.

Listing attributes per object in class trees

Let's code one last extension. As it is, our latest lister includes inherited names but doesn't give any sort of designation of the classes from which the names are acquired. As we saw in the *classtree.py* example near the end of [Chapter 29](#), though, it's straightforward to climb class inheritance trees in code.

The mix-in class in [Example 31-13](#), coded in file *listtree.py*, makes use of this same technique to display attributes grouped by the classes in which they live—it sketches the full *physical class tree*, displaying attributes attached to each object along the way. The reader must still infer attribute inheritance (and we'll address this in the next section's final demo), but this version gives substantially more detail than a simple flat list of attributes, inherited or not.

Example 31-13. *listtree.py*

```
class ListTree:
    """
    Mix-in that returns a __str__ trace of the entire class tree and all
    its objects' attrs at and above the self instance. The display is
    run by print() automatically; use str() to fetch as a string. This:

    -Uses __X pseudoprivate attr names to avoid conflicts with clients
    -Recurses to superclasses explicitly in DLFR (though not MRO) order
    -Uses __dict__ instead of dir() because attrs are listed per object
    -Supports classes with slots: lack of slots here ensures a __dict__
    """

    def __attrnames(self, obj, indent, unders=True):
        spaces = ' ' * (indent + 1)
        result = ''
        for attr in sorted(obj.__dict__):
            if attr.startswith('__') and attr.endswith('__'):
                if unders: result += f'{spaces}{attr}\n'
            else:
                result += f'{spaces}{attr}={getattr(obj, attr)!r}\n'
        return result

    def __listclass(self, aClass, indent):
        dots = '.' * indent
        preamble = (f'\n{dots}'
                    f'<Class {aClass.__name__}'
                    f', address {id(aClass):#x}')

```

```
import testmixin
testmixin.tester(ListTree)      # Test class in this module
```

This class achieves its goal by traversing the inheritance tree—from an instance’s `__class__` to its class, and then from the class’s `__bases__` to all superclasses recursively, scanning each object’s attribute `__dict__` along the way to enumerate attributes. Ultimately, it concatenates each tree portion’s string as the recursion unwinds.

It can take a few moments to understand recursive programs like this, but given the arbitrary shape and depth of class trees, we really have no choice here (apart from explicit stack equivalents of the sorts we met in Chapters 19 and 25, which tend to be no simpler, and which we’ll omit here for space and time). This class is coded to keep its business as explicit as possible, though, to maximize clarity.

To test, run this class’s module file as before; it passes the `ListTree` class to `testmixin.py` of [Example 31-11](#) again, to be mixed in with a subclass in the test function. The file’s tree-sketcher output is as follows:

```
$ python3 listtree.py
<Instance of Sub, address 0x10a45f980:
  _ListTree__visited={}
  data1='code'
  data2='Python'
  data3=3.12

....<Class Sub, address 0x7fb26a448530:
  __doc__
  __init__
  __module__
  method2=<function tester.<locals>.Sub.method2 at 0x10a482ac0>

.....<Class Super, address 0x7fb26a445c00:
  __dict__
  __doc__
  __init__
  __module__
  __weakref__
  method1=<function tester.<locals>.Super.method1 at 0x10a4823e0>

.....<Class object, address 0x10a01b100: (see dir(object))>
.....>

.....<Class ListTree, address 0x7fb26a443d20:
  _ListTree__attrnames=<function ListTree.__attrnames at 0x10a480f40>
  _ListTree__listclass=<function ListTree.__listclass at 0x10a480fe0>
  __dict__
  __doc__
  __module__
  __str__
  __weakref__

.....<Class object, address 0x10a01b100: (see above)>
.....>
....>
>
```

Some points to notice about this example:

- The `__visited` table's name is mangled in the instance's attribute dictionary for *pseudoprivacy*; unless we're very unlucky, this won't clash with other data there. Some of the `lister` class's methods are mangled as well.
- To minimize displays, `__X__` attributes are listed by *name* only, skipping their values. The built-in object class implied above all topmost classes is also singled out to simply refer readers to its `dir` result; object comes with 24 attributes today, which wouldn't be useful to repeat in every display.
- The attributes that were bound methods in the prior version are now plain *functions*. This reflects the fact that this version fetches methods from *classes* instead of the instance—the `getattr` here is run on the current tree object whose `__dict__` is being scanned (`getattr` and `__dict__` indexing are equivalent in this context). Again, class methods are just functions, which are bound only when fetched from an instance.
- To avoid listing a class object more than once, a table records classes *visited* so far. A dictionary works in this role because class objects are hashable and thus may be dictionary keys; a set would work similarly.

On the last point, *cycles* are not generally possible in class inheritance trees—a class must already have been defined to be named as a superclass, and Python raises an exception as it should if you attempt to create a cycle later by `__bases__` changes. The visited mechanism here is still needed, though, to avoid relisting a class twice:

```
>>> class C: pass
>>> class B(C): pass
>>> C.__bases__ = (B,)          # Dark magic
TypeError: a __bases__ item causes an inheritance cycle
```

For more fun, try mixing this class into something more substantial, like the `Button` class of Python's `tkinter` GUI-toolkit module. In general, you'll want to name `ListTree` first (*leftmost*) in a class header, so its `__str__` is picked up; `Button` has one, too, and the leftmost superclass is searched first in multiple inheritance's default:

```
>>> from tkinter import Button
>>> from listtree import ListTree

>>> class ButtonPlus(ListTree, Button): pass          # ListTree's str, not Button's
>>> print(ButtonPlus())
...our class's display...

>>> class ButtonPlus(Button, ListTree): pass          # Mix-in order can matter!
>>> print(ButtonPlus())
.!buttonplus2
```

Order matters in multiple inheritance, though the manual overrides we explored earlier can force the issue:

```
>>> class ButtonPlus(Button, ListTree): __str__ = ListTree.__str__
...our class's display...
```

You might also try running `testmixin.py` of [Example 31-11](#) directly; its self-test code that we shelved earlier runs each of our three `lister` classes in turn, using their module and class name strings—a trivial class factory in action:

```
$ python3 testmixin.py
...all three listers' results...
```

While our tree lister works as planned, it doesn't really follow Python's MRO ordering through class trees with diamonds. In fact, it really just sketches the DFLR order and leaves it to readers to determine from which class a given attribute is inherited. To do better, let's move on to a final example to close out this chapter.

Example: Mapping Attributes to Inheritance Sources

This section wraps up with an example that demos an application for the MRO in programming tools. As coded, the preceding section's tree lister gave the *physical* locations of attributes in a class tree. However, by mapping the list of inherited attributes in a `dir` result to the linear MRO sequence, such tools can more directly associate attributes with the classes from which they are actually *inherited*—also a useful relationship for programmers.

We won't recode our tree lister in full here, but as a first major step, [Example 31-14](#), file `mapattrs.py`, implements tools that can be used to associate attributes with their inheritance source. As an added bonus, its `mapattrs` function demonstrates how inheritance actually searches for attributes in class tree objects—though the MRO is largely automated for us with the built-in `__mro__` class attribute we met earlier.

Example 31-14. `mapattrs.py`

```
"""
Main tool: mapattrs() maps all attributes on or inherited by an
instance to the instance or class from which they are inherited.
Also here: assorted dictionary tools using comprehensions.

Assumes dir() gives all attributes of an instance. To emulate
inheritance, this uses the class's __mro__ tuple, which gives the
MRO search order for classes in Python 3.X. A recursive tree
traversal for the DFLR order of classes is included but unused.
"""

import pprint
def trace(label, X, end='\n'):
    print(f'{label}\n{pprint.pformat(X)}{end}') # Print nicely

def filterdictvals(D, V):
    """
    dict D with entries for value V removed.
    filterdictvals(dict(a=1, b=2, c=1), 1) => {'b': 2}
    """
    return {K: V2 for (K, V2) in D.items() if V2 != V}

def invertdict(D):
    """
    dict D with values changed to keys (grouped by values).
    Values must all be hashable to work as dict/set keys.
    invertdict(dict(a=1, b=2, c=1)) => {1: ['a', 'c'], 2: ['b']}
    """
    def keysof(V):
        return sorted(K for K in D.keys() if D[K] == V)
    return {V: keysof(V) for V in set(D.values())}

def dflr(cls):
    """
    Depth-first left-to-right order of class tree at cls.
```

```

Cycles not possible: Python disallows on __bases__ changes.
"""
here = [cls]
for sup in cls.__bases__:
    here += dflr(sup)
return here

def inheritance(instance):
    """
    Inheritance order sequence: MRO or DFLR.
    DFLR alone is no longer used in Python 3.X.
    """
    if hasattr(instance.__class__, '__mro__'):
        return (instance,) + instance.__class__.__mro__
    else:
        return [instance] + dflr(instance.__class__)

def mapattrs(instance, withobject=False, bysource=False):
    """
    dict with keys giving all inherited attributes of instance,
    with values giving the object that each is inherited from.
    withobject: False=remove object built-in class attributes.
    bysource: True=group result by objects instead of attributes.
    Supports classes with slots that preclude __dict__ in instances.
    """
    attr2obj = {}
    inherits = inheritance(instance)
    for attr in dir(instance):
        for obj in inherits:
            if hasattr(obj, '__dict__') and attr in obj.__dict__: # Slots okay
                attr2obj[attr] = obj
                break

    if not withobject:
        attr2obj = filterdictvals(attr2obj, object)
    return attr2obj if not bysource else invertdict(attr2obj)

if __name__ == '__main__':

    class D:
        attr2 = 'D'
    class C(D):
        attr2 = 'C'
    class B(D):
        attr1 = 'B'
    class A(B, C):
        pass
    I = A()
    I.attr0 = 'I'

    print(f'Py=>{I.attr0=}, {I.attr1=}, {I.attr2=}\n') # Python's search
    trace('INHERITANCE', inheritance(I)) # [Inheritance order]
    trace('ATTRIBUTES', mapattrs(I)) # {Attr => Source}
    trace('SOURCES', mapattrs(I, bysource=True)) # {Source => [Attrs]}

```

This module's main `mapattrs` function uses `dir` to collect all the attributes that an instance inherits. For each, it maps the attribute to its source by scanning the MRO order available in the `__mro__` of the instance's class, searching each object's namespace `__dict__` along the way. The net effect replicates Python's true inheritance search for each attribute accessible from the instance passed in.

This file's self-test code applies its tools to a diamond multiple-inheritance tree similar to those we studied earlier. It uses Python's `pprint` standard-library module to display lists and dictionaries nicely—`pprint.pprint` is its basic call, and its `pformat` returns a print string. Notably, `attr2`, whose value is

given on the first line and whose name appears in later function results, is inherited from class C per the MRO order we've studied:

```
$ python3 mapattr.py
Py=>I.attr0='I', I.attr1='B', I.attr2='C'

INHERITANCE
(<__main__.A object at 0x10cc33e00>,
 <class '__main__.A'>,
 <class '__main__.B'>,
 <class '__main__.C'>,
 <class '__main__.D'>,
 <class 'object'>)
```

```
ATTRIBUTES
{'__dict__': <class '__main__.D'>,
 '__doc__': <class '__main__.A'>,
 '__module__': <class '__main__.A'>,
 '__weakref__': <class '__main__.D'>,
 'attr0': <__main__.A object at 0x10cc33e00>,
 'attr1': <class '__main__.B'>,
 'attr2': <class '__main__.C'>}
```

```
SOURCES
{<__main__.A object at 0x10cc33e00>: ['attr0'],
 <class '__main__.D'>: ['__dict__', '__weakref__'],
 <class '__main__.C'>: ['attr2'],
 <class '__main__.B'>: ['attr1'],
 <class '__main__.A'>: ['__doc__', '__module__']}
```

Although this module was not designed to be a mix-in class itself, listers may index its `mapattr` function's dictionary results to obtain an attribute's source or a source's attributes. Moreover, it's easy to adapt this module's results to be a mix-in by wrapping them in a `__str__`. Here it is listing attributes' sources in the test classes of the prior section's [Example 31-11](#):

```
$ python3
>>> import pprint
>>> from mapattr import mapattr
>>> class ListAttr2Source:
    def __str__(self):
        return pprint.pformat(mapattr(self))

>>> from testmixin import tester
>>> tester(ListAttr2Source)
{'__dict__': <class 'testmixin.tester.<locals>.Super'>,
 '__doc__': <class 'testmixin.tester.<locals>.Sub'>,
 '__init__': <class 'testmixin.tester.<locals>.Sub'>,
 '__module__': <class 'testmixin.tester.<locals>.Sub'>,
 '__str__': <class '__main__.ListAttr2Source'>,
 '__weakref__': <class 'testmixin.tester.<locals>.Super'>,
 'data1': <testmixin.tester.<locals>.Sub object at 0x102a8fa40>,
 'data2': <testmixin.tester.<locals>.Sub object at 0x102a8fa40>,
 'data3': <testmixin.tester.<locals>.Sub object at 0x102a8fa40>,
 'method1': <class 'testmixin.tester.<locals>.Super'>,
 'method2': <class 'testmixin.tester.<locals>.Sub'>}
```

Listing sources' attributes is just as easy (see also Python's docs for pprint options like compact and width):

```
>>> class ListSource2Attr:
    def __str__(self):
        return pprint.pformat(mapattrs(self, bysource=True))

>>> tester(ListSource2Attr)
{<testmixin.tester.<locals>.Sub object at 0x102ad12e0>: ['data1',
                                                         'data2',
                                                         'data3'],

 <class 'testmixin.tester.<locals>.Super'>: ['__dict__',
                                             '__weakref__',
                                             'method1'],

 <class 'testmixin.tester.<locals>.Sub'>: ['__doc__',
                                           '__init__',
                                           '__module__',
                                           'method2'],

 <class '__main__.ListSource2Attr'>: ['__str__']}
```

Study this example's code for more insight. As callouts, notice how it uses `hasattr` to check whether an object has a `__dict__` attribute dictionary before trying to index it. Though rare, some instances may not have a `__dict__` if they use the class extension known as *slots* noted earlier. The prior section's slot story is varied: as mix-ins, `ListTree` and `ListInstance` work as is for classes with slots because their *lack* of slots ensures an instance `__dict__`, but `ListInherited` can fail for slots not yet assigned—findings to be clarified in the next chapter.

Additionally, both slots and other “virtual” instance attributes like *properties* and *descriptors* live at the class instead of the instance and hence may require generic handling—`dir` enumeration, and either `getattr` fetches, tree climbs, or MRO scans. This example and the prior section's listers accommodate this, but unevenly: some such names will be associated with the classes in which their implementations live, not the instance through which they are accessed.

Moreover, no lister can show attribute names dynamically computed in full by methods like `__getattr__` because these names have no physical basis. Classes implementing such dynamic names can also define a `__dir__` method to provide an attribute result list for `dir` calls, but general tools like our listers and mapper cannot depend on this optional and relatively uncommon interface being present.

Finally, all the attribute listers and mappers in this chapter work in full for normal *instances* but don't support *classes*. For the latter, prints run a default display instead of any of the three listers, and `mapattrs` strangely attributes most names to a mystery class called “type.” The lister skips stem from the fact that built-ins like `print` skip the “instance,” as we've noted before. The `mapattrs` oddity reflects the fact that classes acquire names from both their own superclass tree (and MRO), and a secondary tree (and MRO) formed by “metaclasses” that we have yet to meet.

But to understand both the inheritance bifurcation of metaclasses, as well as ethereal attributes like slots, properties, and descriptors, we need to move on to the next chapter.

Other Design-Related Topics

In this chapter, we’ve studied an assortment of design patterns used to combine classes in Python programs, along with the mechanism behind some of them. We’ve really only scratched the surface here in the design patterns domain, though. Elsewhere in this book, you’ll find coverage of other design-related topics, such as:

- *Abstract superclasses* (Chapter 29)
- *Decorators* (Chapters 32 and 39)
- *Type subclasses* (Chapter 32)
- *Static and class methods* (Chapter 32)
- *Managed attributes* (Chapters 32 and 38)
- *Metaclasses* (Chapters 32 and 40)

For even more details on design patterns, this book must delegate to other resources on OOP at large. Although patterns are important in OOP work and are often more natural in Python than other languages, they are not specific to Python itself and a subject that’s often best acquired by experience.

Chapter Summary

In this chapter, we sampled common ways to use and combine classes to optimize their reusability and factoring benefits—what are usually considered design issues, which are often independent of any particular programming language (though Python can make them easier to implement). We studied *inheritance* (acquiring behavior from other classes), *composition* (controlling embedded objects), and *delegation* (wrapping objects in proxy classes), as well as the related topics of pseudoprivate attributes, bound methods, factories, multiple inheritance, and the MRO.

The next chapter concludes our look at classes and OOP by surveying class-related topics that are more esoteric than most of what we’ve already seen. Some of its material may be of more interest to tool writers than application programmers, but it still merits a review by most people who will do OOP in Python—if not for your code, then for others’ code you may need to understand and reuse. First, though, here’s another quick chapter quiz to review.

Test Your Knowledge: Quiz

1. What is multiple inheritance?
2. What is composition?
3. What is delegation?
4. What are bound methods?
5. What are pseudoprivate attributes used for?
6. How does the MRO inheritance search order differ from DFLR?

Test Your Knowledge: Answers

1. *Multiple inheritance* occurs when a class inherits from more than one superclass; it's useful for mixing together multiple packages of class-based code. The left-to-right order in `class` statement headers determines the general order of attribute searches, and the MRO specializes search for diamonds with common superclasses.
2. *Composition* is a technique whereby a controller class embeds and directs a number of objects and provides an interface all its own; it's a way to build up larger structures with classes.
3. *Delegation* involves wrapping an object in a proxy class, which adds extra behavior and passes other operations to the wrapped object. The proxy generally retains the interface of the wrapped object.
4. *Bound methods* combine an instance and a method function; you can call them without passing in an instance object explicitly because the original instance is still available in the instance+function pair.
5. *Pseudoprivate attributes*, whose names begin but do not end with two leading underscores (e.g., `__x`), are used to localize names to the enclosing class. This includes both class attributes, like methods defined inside the `class` statement, and `self` instance attributes assigned inside the class's methods. Such names are expanded to include the class name, which makes them generally unique among all classes in an inheritance tree.
6. The MRO selects same-named attributes in a *lower* subclass over those in a higher common superclass in multiple-inheritance “diamond” trees—effectively searching across before up in this specific case. The DFLR is otherwise the same; in fact, the MRO is defined by *starting* with the DFLR order and then *removing* all but the last (rightmost) appearances of classes that are visited more than once. This differs from DFLR only when there are duplicates, which arise only in diamonds that have common superclasses. That said, the built-in `object` makes every multiple-inheritance tree a diamond, so this is a common, if implicit, occurrence.

Class Odds and Ends

This chapter concludes our look at OOP in Python by presenting a collage of more advanced class topics. We will survey customizing built-in types, the relationship of classes and types, attribute tools like slots and properties, the special-case static and class methods, decorators and metaclasses, and the super call's complete story. Some of these are introduced here but resumed by focused chapters in this book's [Part VIII, “Advanced Topics”](#).

As we've seen, Python's OOP model is, at its core, relatively simple, and some of the topics presented in this chapter are so advanced and optional that you may not encounter them very often in your Python applications-programming career. In the interest of completeness, though—and because you never know when an “advanced” topic may crop up in code you use—we'll round out our discussion of classes with a brief look at these advanced tools for OOP work.

As usual, because this is the last chapter in this part of the book, it ends with a section on class-related “gotchas” and a set of lab exercises for this part to help cement the ideas we've studied here. Beyond these exercises, studying larger OOP Python projects or starting some of your own is heartily recommended as a supplement to this book. As with much in life and computing, the benefits of OOP tend to become more apparent with practice.



Blast from the past: Python 3.X launched with a mandatory “new-style” class model that could be enabled in 2.X as an option; 2.X's own model was dubbed “classic.” At least in terms of its OOP support, new-style classes transformed Python into a different language altogether—one that borrows much more from, and is often as complex as, other languages in this domain. The last chapter's MRO and most topics in this chapter were part of this package. Because this book is now focused on 3.X only, the term “new style” is moot and unused here—all its classes qualify.

Extending Built-in Object Types

Besides implementing new kinds of objects, classes are sometimes used to extend the functionality of Python's built-in object types to support more exotic data structures. For instance, to add *queue* insert and delete methods to lists, you can code classes that wrap (embed) a list object and augment it with insert and delete methods that process the list specially, using the delegation technique we studied in

Chapter 31. You can also use simple inheritance to customize built-in types for such custom roles. The next two sections show both techniques in action.

Extending Types by Embedding

Do you remember those set functions we wrote in Chapters 16 and 18? Here's what they look like brought back to life as a Python class. **Example 32-1** (file *setwrapper.py*) implements a new set object type by moving set functions to methods and adding some basic operator overloading. For the most part, this class just wraps a Python list with extra set operations. But because it's a class, it also supports multiple instances and customization by inheritance in subclasses. Unlike our earlier functions, using classes here allows us to make multiple self-contained set objects with preset data and behavior rather than passing lists into functions manually.

Example 32-1. *setwrapper.py*

```
class Set:
    def __init__(self, value = []):      # Constructor
        self.data = []                 # Manages a list
        self.concat(value)              # Removes duplicates

    def intersect(self, other):           # other is any iterable
        res = []                       # self is the subject
        for x in self.data:
            if x in other:               # Pick common items
                res.append(x)
        return Set(res)                 # Return a new Set

    def union(self, other):               # other is any iterable
        res = self.data[:]              # Copy of my list
        for x in other:                 # Add items in other
            if not x in res:
                res.append(x)
        return Set(res)

    def concat(self, value):              # value: list, Set...
        for x in value:                 # Removes duplicates
            if not x in self.data:
                self.data.append(x)

    def __len__(self):                   return len(self.data)           # len(self), if self
    def __getitem__(self, key):          return self.data[key]           # self[i], self[i:j]
    def __and__(self, other):            return self.intersect(other)     # self & other
    def __or__(self, other):             return self.union(other)        # self | other
    def __repr__(self):                  return f'Set({self.data!r})'     # print(self),...
    def __iter__(self):                  return iter(self.data)         # for x in self,...
```

To use this class, we make instances, call methods, and run defined operators as usual:

```
$ python3
>>> from setwrapper import Set
>>> x = Set([1, 3, 5, 7, 3])
>>> x.union(Set([1, 4, 7]))
Set([1, 3, 5, 7, 4])
>>> x | Set([1, 4, 6, 4])
Set([1, 3, 5, 7, 4, 6])
```

Overloading operations such as indexing and iteration also enables instances of our Set class to often masquerade as real lists. Because you will interact with and extend this class in an exercise at the end of this chapter, we'll put this code on the back burner until its solution in [Appendix B](#).

Extending Types by Subclassing

While the prior section's embedding works, Python's built-in types can also be subclassed directly. In fact, type-conversion functions such as `list`, `str`, `dict`, and `tuple` are really built-in type names; although transparent to your script, a type-conversion call (e.g., `list('text')`) is really an invocation of a type's constructor.

This allows you to customize or extend the behavior of built-in types with user-defined class statements: simply subclass the type names to customize them. Instances of your type subclasses can generally be used anywhere that the original built-in type can appear. For example, suppose you have trouble getting used to the fact that Python list offsets begin at 0 instead of 1. Not to worry—you can always code your own subclass that customizes this core behavior of lists, and [Example 32-2](#) shows how.

Example 32-2. typesubclass.py

```
"""
Subclass built-in list type/class.
Map 1..N to 0..N-1, call back to built-in version.
"""

class MyList(list):
    def __getitem__(self, offset):
        print(f'indexing {self} at {offset}>')
        return list.__getitem__(self, offset - 1)

if __name__ == '__main__':
    print(list('abc'))
    x = MyList('abc')           # __init__ inherited from list
    print(x)                   # __str__/__repr__ inherited from list

    print(x[1])                # MyList.__getitem__
    print(x[3])                # Customizes list superclass method

    x.append('hack!'); print(x) # Attributes from list superclass
    x.reverse();               print(x)
```

In this file, the `MyList` subclass extends the built-in list's `__getitem__` indexing method only, to map indexes 1 to N back to the required 0 to N-1. Really, all it does is decrement the submitted index and call back to the superclass's version of indexing, but it's enough to do the trick:

```
$ python3 typesubclass.py
['a', 'b', 'c']
['a', 'b', 'c']
<indexing ['a', 'b', 'c'] at 1>
a
<indexing ['a', 'b', 'c'] at 3>
c
['a', 'b', 'c', 'hack!']
['hack!', 'c', 'b', 'a']
```

This output also includes tracing text the class prints on indexing. Of course, whether changing indexing this way is a good idea, in general, is *another issue*—users of your `MyList` class may very well be

confused by such a core departure from Python sequence behavior. The ability to customize built-in types this way can be a powerful asset, though.

For instance, this coding pattern gives rise to an alternative way to code a set—as a subclass of the built-in list type rather than a standalone class that manages an embedded list object, as shown in the prior section. As discussed in [Chapter 5](#), Python today comes with a powerful built-in set object, along with literal and comprehension syntax for making new sets. Coding one yourself, though, is still a great way to learn about type subclassing in general.

The code in [Example 32-3](#), file *setsubclass.py*, customizes lists to add just methods and operators related to set processing. Because all other behavior is inherited from the built-in list superclass, this makes for a shorter and simpler alternative—everything not defined here is routed to list directly.

Example 32-3. *setsubclass.py*

```
class Set(list):
    def __init__(self, value = []):      # Constructor
        list.__init__(self)            # Customizes list
        self.concat(value)              # Copies mutable defaults

    def intersect(self, other):          # other is any iterable
        res = []                       # self is the subject
        for x in self:
            if x in other:              # Pick common items
                res.append(x)
        return Set(res)                # Return a new Set

    def union(self, other):              # other is any iterable
        res = Set(self)                # Copy me and my list
        res.concat(other)
        return res

    def concat(self, value):             # value: list, Set, etc.
        for x in value:                # Removes duplicates
            if not x in self:
                self.append(x)

    def __and__(self, other): return self.intersect(other)
    def __or__(self, other):  return self.union(other)
    def __repr__(self):       return f'Set({list.__repr__(self)})'

if __name__ == '__main__':
    x = Set([1, 3, 5, 7])
    y = Set([2, 1, 4, 5, 6])
    print(x, y, len(x))
    print(x.intersect(y), y.union(x))
    print(x & y, x | y)
    x.reverse(); print(x)
```

Here is the output of the self-test code at the end of this file. Because subclassing core types is a somewhat advanced feature with a limited audience, we'll end this topic here, but you're invited to trace through these results in the code to study its behavior:

```
$ python3 setsubclass.py
Set([1, 3, 5, 7]) Set([2, 1, 4, 5, 6]) 4
Set([1, 5]) Set([2, 1, 4, 5, 6, 3, 7])
Set([1, 5]) Set([1, 3, 5, 7, 2, 4, 6])
Set([7, 5, 3, 1])
```

Subtleties: some inherited list operations may introduce duplicates to our Set, and there are more efficient ways to implement sets with dictionaries in Python, which replace the nested linear search scans in the set implementations shown here with more direct dictionary index operations (hashing) and so run much quicker. If you're interested in sets, also take another look at the set object type we explored in [Chapter 5](#); this type provides extensive set operations as built-in tools. Set implementations are fun to experiment with but not strictly required in Python today.

More important here is the question of why we can subclass built-in types like list at all. The next section solves the mystery—at least as much as this chapter can.

The Python Object Model

The reason we could subclass built-in types in the prior section is that types and classes are largely one and the same—a unification that came with the “new-style” model alluded to at the start of this chapter. For built-ins, some instances can uniquely be coded with literal syntax like [], 'lp6e', and 3.12 instead of class calls like list(), str(), and float(), but they are instances of a class, nonetheless.

In fact, built-in types and user-defined classes are both classes and are both themselves instances of the built-in type class. The type object generates classes as its instances, classes generate instances of themselves, and classes are really just user-defined types. And on top of all this, the built-in object class provides defaults for every object.

Classes Are Types Are Classes

While you probably shouldn't ponder the preceding definitions before operating heavy machinery, it's easy to see all this in code. The type built-in with one argument returns any object's type, which is normally the same as the object's __class__, and isinstance checks whether an object inherits from another. Here's the story for user-defined classes (a.k.a. types):

```
>>> class Hack: pass           # A humble user-defined class
>>> I = Hack()                 # Make an instance by calling the class

>>> type(I)                    # Type is the user-defined class of origin
<class '__main__.Hack'>

>>> type(Hack)                 # User-defined classes are instances of type
<class 'type'>

>>> I.__class__, Hack.__class__
(<class '__main__.Hack'>, <class 'type'>)

>>> isinstance(I, object), isinstance(Hack, object)
(True, True)
```

This works the same for built-in types (a.k.a. classes), but there is also literal syntax for generating instances:

```
>>> I = 'hack'                 # Make an instance by literal syntax, or str()

>>> type(I)                    # Built-in objects are instances of classes
<class 'str'>

>>> type(str)                  # Built-in classes are instances of type
<class 'type'>

>>> I.__class__, str.__class__
```

```
(<class 'str'>, <class 'type'>)

>>> isinstance(I, object), isinstance(str, object)
(True, True)
```

In fact, `type` itself reports in as a class, though it has no type but itself—circularly capping the chain:

```
>>> type(type)                # The type class ends the chain
<class 'type'>
>>> type(type(type))          # Hmm...the top of the chain
<class 'type'>
```

This model may seem academic (and to some extent is), but it allows us to specialize built-in types with normal user-defined classes and bears on type-testing code: you must know what a type is to test it accurately.

Some Instances Are More Equal Than Others

It's tempting to simply take away from the foregoing that classes and types are the same, but this story is richer than that may imply: the instances we make from these objects diverge in both functionality and inheritance.

To truly understand how, we have to briefly factor in *metaclasses*—classes that generate other classes. The type built-in itself *is* a metaclass and may be *customized* with user-defined subclasses. These subclasses are coded with normal `class` statements, selected with special syntax in `class` headers, and designed to play metaclass roles, but they won't be covered in full until [Chapter 40](#).

In brief, though, the complete relationship between instances, classes, and types is as follows:

- *Instances* are created from classes—both built-in and user-defined.
- *Classes* themselves are created from the built-in `type` class or one of its subclasses.
- The object built-in class is a superclass to every object—instance, class, or both.

Although everything is ultimately an “instance” in Python, there are two fundamentally different *kinds* of instances, and conflating these only serves to mask the true complexity of the model. It doesn't help to distinguish these as instances of *user-defined* classes or not: metaclasses may be user-defined classes too. Nor is this about being a *subclass* of a type: the real fork in this model is that classes are *created* from a type class specially.

As you'll learn in full later, classes define their types with optional `metaclass` syntax that defaults to `type` if omitted, but other instances define their types by the class calls or literal syntax we've used so far:

```
class C(metaclass=Meta): ...    # Class creation: metaclass defaults to "type"

I = C(...)                    # Instance creation: user-defined classes
X = [1, 2]                    # Instance creation: built-in classes
```

While both produce instances in some sense, these different syntaxes create very different kinds of instances—*class* and *nonclass*—with fundamentally different behaviors:

Nonclass instances do not make instances

Classes are created from `type` (or another metaclass), similar to the way instances are created from classes. Once created, though, the analogy fails: *classes* create instances of their own, but *nonclass* instances do not.

Classes have an extra inheritance search

There are really *two* inheritance trees and searches in Python, which are distinct but not entirely disjoint. The secondary tree is formed by `type` and its subclasses and is searched only for *classes*, not *nonclass* instances.

In other words, nonclass instances seal off the instantiation chain, and inheritance differs for nonclass instances and classes themselves—even though the latter are also instances of `type`. All of this boils down to different creation syntax that makes different kinds of objects, which are often confusingly lumped together as “instances”:

```
>>> class C: pass                                # A type instance
>>> I = C()                                       # A nonclass instance
>>> isinstance(I, type), isinstance(C, type)     # Only classes are types
(False, True)
```

While types and classes may be synonymous, the instances we create from them vary per creation code.

The Inheritance Bifurcation

Though we can’t get into full details here, the inheritance search used for *classes* (a.k.a. types) differs from what we’ve seen so far and may be their most profound distinction. In short, inheritance is always based on the MRO (method resolution order) we studied in [“Multiple Inheritance and the MRO” on page 796](#), but varies as follows:

Nonclass inheritance

As we’ve seen, inheritance run on a *nonclass* instance searches the `__dict__` attributes of instance, class, and superclasses, per the MRO order we studied in the prior chapter. This works by first checking the instance, then following the instance’s `__class__` to its class, and finally following each class’s `__bases__` to superclasses. Technically, `__bases__` are used to make an `__mro__` at `__class__`, which inheritance scans.

Class inheritance

Inheritance run on a *class* directly, though, first searches the `__dict__` of the class and all its supers available from `__bases__` as usual, but then *also* searches the separate class tree formed by the type class and its *metaclass* subclasses. The second part of this works by following the class’s own `__class__` to its type class tree and using `__bases__` and MROs there, too—but only as a last resort and only for inheritance run on classes.

In fact, if you know where to look, you can inspect the inheritance sources that differ for nonclass instances like `I` and classes like `C` in the prior example—though the underscores and displays aren’t pretty:

```
>>> isinstance(I, C), type(I)
(True, <class '__main__.C'>)

>>> I, I.__class__, I.__class__.__bases__
(<__main__.C object at 0x101265d60>, <class '__main__.C'>, (<class 'object'>,,))

>>> C, C.__bases__, C.__class__, C.__class__.__bases__
(<class '__main__.C'>, (<class 'object'>,,), <class 'type'>, (<class 'object'>,,))
```

But due to the way MROs are computed from `__bases__` and scanned, it’s more accurate to think of inheritance’s different search orders for nonclass instances and classes as follows—where each `__mro__` is a *flattened tree*:

```
>>> I, I.__class__.__mro__
(<__main__.C object at 0x101265d60>, (<class '__main__.C'>, <class 'object'>))

>>> C.__mro__, C.__class__.__mro__
((<class '__main__.C'>, <class 'object'>), (<class 'type'>, <class 'object'>))
```

And because each item's `__dict__` is checked, the ordered set of candidates searched by inheritance for nonclass instances `I` and classes `C` is ultimately and respectively as follows—with *two* MRO scans of flattened trees for classes only, and ignoring the fact that some kinds of *descriptors*, introduced ahead, take precedence in both trees as you'll learn in [Chapter 40](#):

```
[I.__dict__] + [x.__dict__ for x in I.__class__.__mro__]

[x.__dict__ for x in C.__mro__] + [x.__dict__ for x in C.__class__.__mro__]
```

Wait—there's a *second* tree in inheritance? Well, yes, though it doesn't come into play in the vast majority of application code. The type/metaclass tree is used in advanced class-management roles and, even then, is often limited to class customization at class creation time.

Still, this secondary tree, along with the descriptors' special cases omitted here, bifurcates and convolutes the inheritance story, especially compared to its prior forms. It also explains why some class attributes like `__bases__` are not inherited by nonclass instances—they're located in the secondary tree (i.e., MRO) searched only for classes:

```
>>> '__bases__' in I.__dict__           # Not in instance
False
>>> '__bases__' in C.__dict__          # Not in instance's class
False
>>> '__bases__' in C.__class__.__dict__ # In instance's class's class
True
```

Because of the two-tree inheritance model, such names inherited by classes are not inherited by their instances:

```
>>> C.__bases__                        # Instance does not inherit!
(<class 'object'>,)
>>> I.__bases__
AttributeError: 'C' object has no attribute '__bases__'. Did you mean: '__class__'?
```

The Metaclass/Class Dichotomy

So, where does this odd tale of type/class unification leave us? Types indeed behave as classes, and this allows us to extend them with normal class syntax in both the primary and metaclass trees. But it also comes with noticeable *seams*, including special-case syntax for class instantiation, an extra type-tree search for classes only, two very different kinds of instances, and unique semantics for metaclasses that customize types (a.k.a. classes), which we'll uncover later.

In fact, a reasonable argument can be made that the *type/class* dichotomy of earlier Pythons may simply have morphed into one of *metaclass/class*—which trades a straightforward distinction for all the seams just enumerated and muddles inheritance and the fundamental meaning of *names* in Python everywhere to support what in the end is a very rare use case. As usual, the net merit of the morph is yours to weigh.

To be fair, some of the widespread confusion this model has spawned may stem from `type` itself: it's overloaded to either *return* a sole argument's type, or *generate* a new instance of itself for multiple arguments—just like other constructors and equivalent to what a `class` statement does to make a class object:

```

type(object)                # Fetch object type
type(classname, superclasses, attributedict) # Make a class/type

```

The first of these roles might have been better named “typeof,” but the second will have to await the metaclass preview later in this chapter and the extended coverage in [Chapter 40](#).

And One “object” to Rule Them All

To round out this topic, keep in mind that because topmost classes inherit from the built-in class `object`, every object *derives* (i.e., inherits) from it, whether directly or through a superclass—and whether you code `object` or not:

```

>>> class C: pass
>>> class D(object): pass

>>> dir(C) == dir(D)
True
>>> C.__bases__, D.__bases__
((<class 'object'>,), (<class 'object'>,,))

```

In fact, the `type` class inherits from the `object` class, and `object` inherits from `type`, even though the two are different objects—a circular relationship that crowns the object model and may make your cranium catch fire (to avoid combustion, keep in mind that `isinstance` is true for *either* a subclass relationship or creation source, though this is based on inheritance through the secondary type-class tree for classes):

```

>>> type is object
False
>>> type(type), type(object)
(<class 'type'>, <class 'type'>)
>>> isinstance(type, object), isinstance(object, type)
(True, True)
>>> type.__bases__, object.__bases__
((<class 'object'>,), ())

```

Strange though it may seem, this has a number of practical consequences. For one thing, it means that we sometimes must be aware of the method defaults that come with the implicit (or explicit) `object` root class. As we noted in earlier chapters, for instance, the `object` class comes with a `__repr__` for display:

```

>>> class C: pass                # All classes inherit object defaults
>>> X = C()
>>> X.__repr__
<method-wrapper '__repr__' of C object at 0x1091a7920>

```

For another, this also allows us to write code that can safely assume and use an `object` superclass. As an example, we can rely on it to be a call-chain “anchor” in some super built-in roles described ahead and can reroute method calls to it from attribute-interceptor methods to invoke higher default behavior. Per [Chapter 30](#):

```

object.__setattr__(self, attr, value)

```

We’ll code examples of such rerouting later in the book; for now, let’s move on to something a bit more tangible and our next topic in this OOP jamboree.

Advanced Attribute Tools

Along with the normal class and instance attributes we've been using so far, Python's OOP support includes attribute tools of narrower scope—*slots*, *properties*, *descriptors*, and more. Slots, for example, are an optimization option, and properties and descriptors allow classes to augment access. None of these tools are required, but as for most topics in this chapter, all are fair game in Python code you may someday use. Most of these tools get extended coverage in [Chapter 38](#), but slots get full coverage here, and others are presented in abbreviated form.

Slots: Attribute Declarations

First off, we've noted the implications of *slots* several times in this part of the book. In short, by assigning an iterable of attribute name strings to a special `__slots__` class attribute, we can enable a class to both limit the set of legal attributes that instances of the class will have and optimize memory usage and possibly program speed. As you'll find, though, slots should be used only in applications that clearly warrant the added complexity. They will complicate your code, may complicate or break code you may use, and rigidly require universal deployment to be effective.

Slot basics

To declare slots, assign an iterable (e.g., list) of string names to the special `__slots__` variable and attribute at the top level of a class statement: only those names in `__slots__` can be assigned as instance attributes. This doesn't change the way these attributes work in general, though; like all names in Python, instance attribute names must always be assigned before they can be referenced, even if they're listed in `__slots__`. Here are the basics:

```
>>> class Limiter(object):
    __slots__ = ['age', 'name', 'job']           # Slots "declaration"

>>> I = Limiter()
>>> I.age                                     # Must assign before use
AttributeError: 'Limiter' object has no attribute 'age'

>>> I.age = 40                               # Looks like instance data
>>> I.age
40

>>> I.ape = 1000                             # Fails: not in __slots__
AttributeError: 'Limiter' object has no attribute 'ape'
```

This feature is advertised as both a way to catch typo errors like this (assignments to illegal attribute names not in `__slots__` are detected instead of silently assigned), as well as an optimization mechanism that saves memory.

Allocating a namespace dictionary for every instance object can be expensive in terms of *memory* if many instances are created and only a few attributes are required. To save space, instead of allocating a dictionary for each instance, Python reserves just enough space in each *instance* to hold a value for each slot attribute, along with inherited attributes in the common *class* to manage slot access. This might additionally *speed* execution, though this benefit may vary per program, platform, and Python version (spoiler: the speedup is trivial today, as we'll prove ahead).

You shouldn't normally use slots

Slots are a fairly major break with Python's core dynamic nature, which dictates that any name may be created by assignment (and frankly, tend to appeal most to people with backgrounds in draconian languages). In fact, they partly imitate C++ for efficiency at the expense of flexibility and even have the potential to *break* some programs.

As you'll see, slots also come with a plethora of special-case usage *rules*. Per Python's own manual, they should *not* be used except in clearly warranted cases—they are difficult to deploy correctly, complicate your code badly, and are best limited to very rare memory-critical programs that produce an extremely large numbers of instances.

In other words, this is yet another feature that should be used only if clearly justified. Unfortunately, slots seem to be showing up in Python code much more often than they should; their obscurity seems to be a draw in itself. Slots are actually used by Python, unlike *type hinting*, their declaration cousin of [Chapter 6](#), but they are similarly paradoxical and restrictive. As usual, knowledge is your best ally in such things, so let's take a deeper look here.

Slots and namespace dictionaries

Potential benefits aside, slots can complicate a class model—and code that relies on it—substantially. In fact, some instances with slots may not have a `__dict__` attribute namespace dictionary at all, and others will have data attributes that this dictionary does not include. To be clear, this is a *major incompatibility* with the traditional class model—one that can impact any code that accesses attributes generically and may even cause some to fail altogether.

For instance, programs that list or access instance attributes by name string may need to use more storage-neutral interfaces than `__dict__` if slots may be used. Because an instance's data may include class-level names such as slots—either in *addition* to or *instead* of namespace dictionary storage—both attribute sources may need to be queried for completeness, and some roles may be rendered impossible.

Let's see what this means in terms of code and explore more about slots along the way. First off, when slots are used, instances do not normally have an attribute dictionary—instead, Python uses the class *descriptors* feature introduced ahead to allocate and manage space reserved for slot attributes in the instance:

```
>>> class C:
    __slots__ = ['a', 'b']           # __slots__ means no __dict__ by default

>>> I = C()
>>> I.a = 1
>>> I.a
1
>>> I.__dict__
AttributeError: 'C' object has no attribute '__dict__'. Did you mean: '__dir__'?
```

However, we can still fetch and set slot-based attributes by name string using storage-neutral tools such as `getattr` and `setattr` (which look beyond the instance `__dict__` and thus include class-level names like slots) and list them with `dir` (which collects all inherited names of any kind throughout a class tree):

```
>>> getattr(I, 'a')
1
>>> setattr(I, 'b', 2)           # But getattr() and setattr() still work
>>> I.b
2
```

```
>>> 'a' in dir(I)                # And dir() finds slot attributes too
True
>>> 'b' in dir(I)                # Though __dict__ access will fail
True
>>> I.__dict__
AttributeError: 'C' object has no attribute '__dict__'. Did you mean: '__dir__'?
```

Also keep in mind that without an attribute namespace dictionary, it's not possible to assign *new* names to instances that are not names in the slots list:

```
>>> class D:
    __slots__ = ['a', 'b']
    def __init__(self):
        self.d = 4                # Cannot add new names if no __dict__

>>> I = D()
AttributeError: 'D' object has no attribute 'd'
```

We can still accommodate extra attributes, though, by including `__dict__` explicitly in `__slots__` in order to create an attribute namespace dictionary in *addition* to slots:

```
>>> class D:
    __slots__ = ['a', 'b', '__dict__'] # Name __dict__ to include one too
    c = 3                             # Class attrs work normally
    def __init__(self):
        self.d = 4                   # d stored in __dict__, a is a slot

>>> I = D()
>>> I.d
4
>>> I.c
3
>>> I.a                # All instance attrs undefined until assigned
AttributeError: 'D' object has no attribute 'a'
>>> I.a = 1
>>> I.b = 2
```

In this case, *both* storage mechanisms are used. This renders `__dict__` too limited for code that wishes to treat slots as instance data, but generic tools such as `getattr` still allow us to process both storage forms as a single set of attributes:

```
>>> I.__dict__          # Some objects have both __dict__ and slot names
{'d': 4}               # getattr() can fetch either type of attr
>>> I.__slots__
['a', 'b', '__dict__']
>>> getattr(I, 'a'), getattr(I, 'c'), getattr(I, 'd')    # Fetches all 3 forms
(1, 3, 4)
```

Because `dir` also returns all *inherited* attributes, though, it might be too broad in some contexts; it also includes class-level methods and even all object defaults. Code that wishes to list *just* instance attributes may, in principle, still need to allow for both storage forms explicitly. We might at first naively code this as follows:

```
>>> for attr in list(I.__dict__) + I.__slots__:
    print(attr, '=>', getattr(I, attr))                # Wrong...
```

Since either can be omitted, we may more correctly code this as follows, using `getattr` to allow for defaults—a noble but nonetheless inaccurate approach, as the next section will explain:

```
>>> for attr in list(getattr(I, '__dict__', [])) + getattr(I, '__slots__', []):
    print(attr, '=>', getattr(I, attr))

d => 4
a => 1
b => 2
__dict__ => {'d': 4}
```

Less wrong...

Multiple `__slot__` lists in superclasses

The preceding code works in this specific case, but in general, it's not entirely accurate. Specifically, this code addresses only slot names in the *lowest* `__slots__` attribute inherited by an instance, but slot lists may appear more than once in a class tree. That is, a name's absence in the lowest `__slots__` list does not preclude its existence in a higher `__slots__`. Because slot names become class-level attributes, instances acquire the *union* of all slot names anywhere in the tree by the normal inheritance rule:

```
>>> class E:
    __slots__ = ['c', 'd']          # Superclass has slots

>>> class D(E):
    __slots__ = ['a', '__dict__']  # But so does its subclass

>>> I = D()                        # The instance gets the union of each
>>> dir(I)
[...names omitted..., 'a', 'c', 'd']
>>> I.a = 1; I.b = 2; I.c = 3      # slots: a, c, __dict__: b
>>> I.a, I.c
(1, 3)
```

But inspecting just the inherited slots list won't pick up slots defined *higher* in a class tree:

```
>>> E.__slots__                    # But __slots__ not concatenated
['c', 'd']
>>> D.__slots__
['a', '__dict__']
>>> I.__slots__                    # Instance inherits *lowest* __slots__
['a', '__dict__']
>>> I.__dict__                     # And has its own attr dict
{'b': 2}

>>> for attr in list(getattr(I, '__dict__', [])) + getattr(I, '__slots__', []):
    print(attr, '=>', getattr(I, attr))

b => 2
a => 1
__dict__ => {'b': 2}

>>> dir(I)                         # But dir() includes all slot names
[...names omitted..., 'a', 'b', 'c', 'd']
```

In other words, in terms of listing instance attributes generically, one `__slots__` isn't always enough—they are potentially subject to the full inheritance search procedure. If multiple classes in a class tree may have their own `__slots__` attributes, tools must develop other policies for listing attributes—as the next section explains.

Handling slots and other “virtual” attributes generically

The prior chapter concluded with a brief summary of the slots policies of its attribute lister tools—a prime example of why generic programs may need to care about slots. Such tools that attempt to list instance data attributes generically must account for slots and perhaps other such “virtual” instance attributes like *properties* and *descriptors* introduced ahead—names that similarly reside in classes but may provide attribute values for instances on request. Slots are the most data-centric of these but are representative of a larger category.

Such attributes require inclusive approaches, special handling, or general avoidance—the latter of which becomes unsatisfactory as soon as any programmer uses slots in subject code. Really, class-level instance attributes like slots probably necessitate a redefinition of the term *instance data*—as locally stored attributes, the union of all inherited attributes, or some subset thereof.

For example, some programs might classify slot names as attributes of *classes* instead of instances; these attributes do not exist in instance namespace dictionaries, after all. Alternatively, as shown earlier, programs can be more inclusive by relying on `dir` to fetch all inherited attribute names and `getattr` to fetch their corresponding values—without regard to their physical location or implementation. If you must support slots as instance data, this may be the most robust way to proceed:

```
>>> class Slotful:
    __slots__ = ['a', 'b', '__dict__']
    def __init__(self, data):
        self.c = data

>>> I = Slotful(3)
>>> I.a, I.b = 1, 2
>>> I.a, I.b, I.c                                # Normal attribute fetch
(1, 2, 3)

>>> I.__dict__                                    # Both __dict__ and slots storage
{'c': 3}
>>> [x for x in dir(I) if not x.startswith('__')]
['a', 'b', 'c']

>>> I.__dict__['c']                               # __dict__ is only one attr source
3
>>> getattr(I, 'c'), getattr(I, 'a')              # dir+getattr is broader than __dict__
(3, 1)                                           # applies to slots, properties, descrip

>>> for a in (x for x in dir(I) if not x.startswith('__')):
    print(a, '=>', getattr(I, a))

a 1
b 2
c 3
```

Under this `dir/getattr` model, you can still map attributes to their inheritance sources and filter them more selectively by source or type, if needed, by scanning the *MRO*—as we did in the prior chapter’s *mapattrs.py* (Example 31-14). As a bonus, such tools and policies for handling slots will potentially apply automatically to properties and descriptors too, though these attributes are more explicitly computed values, and less obviously instance-related data than slots.

Also keep in mind that this is not just a tools issue. Class-based instance attributes like slots also impact the traditional coding of the `__setattr__` operator-overloading method we met in Chapter 30. Because slots and some other attributes are not stored in the instance `__dict__`, and may even imply its *absence*, classes must instead generally run attribute assignments by rerouting them to the object superclass.

Slot usage rules

Slot declarations can appear in multiple classes in a class tree, but when they do, they are subject to a number of constraints that are somewhat difficult to rationalize unless you understand the implementation of slots as class-level *descriptors* for each slot name that are inherited by the instances in which the managed space is reserved (again, you'll meet descriptors briefly ahead). Here are the main constraints that slots impose:

- **Slots in subs are pointless when absent in supers.** If a subclass inherits from a superclass without a `__slots__`, the instance `__dict__` attribute created for the superclass will always be accessible, making a `__slots__` in the subclass largely pointless. The subclass still manages its slots but doesn't compute their values in any way and doesn't avoid a dictionary—the main reason to use slots.
- **Slots in supers are pointless when absent in subs.** Similarly, because the meaning of a `__slots__` declaration is limited to the class in which it appears, subclasses will produce an instance `__dict__` if they do not define a `__slots__`, rendering a `__slots__` in a superclass largely pointless.
- **Redefinition renders super slots pointless.** If a class defines the same slot name as a superclass, its redefinition hides the slot in the superclass per normal inheritance. You can access the version of the name defined by the superclass slot only by fetching its descriptor directly from the superclass.
- **Slots prevent class-level defaults.** Because slots are implemented as class-level descriptors (along with per-instance space), you cannot use class attributes of the same name to provide defaults as you can for normal instance attributes: assigning the same name in the class overwrites the slot descriptor.
- **Slots cannot be combined in multiple inheritance.** Multiple inheritance cannot be used if more than one of the classes mixed together have nonempty slots lists—even if their slots define the same names. You'll get an error when running the class that does the mixing. Empty slots lists allow the mixer to define slots or not, as desired.
- **Slots can impact `__dict__`.** As shown earlier, `__slots__` preclude both an instance `__dict__` and assigning names not listed, unless `__dict__` is listed explicitly too. Slots similarly preclude a `__weakref__` attribute used to support instance “weak references” covered briefly in [Chapter 6](#), but these are rare enough to soft-pedal here.

We've already seen the last of these in action. It's easy to demonstrate how the new rules here translate to actual code—most crucially, a namespace dictionary is created when any class in a tree omits slots, thereby negating the memory optimization benefit but also supporting classes that require a `__dict__` when mixed in with others:

```
>>> class C: pass
>>> class D(C): __slots__ = ['a']
>>> I = D()
>>> I.a = 1; I.b = 2
>>> I.__dict__
{'b': 2}
>>> D.__dict__.keys()
dict_keys(['__slots__', 'a', ...])

# Bullet 1: slots in sub but not super
# Makes instance dict for nonslots
# But slot name still managed in class

>>> class C: __slots__ = ['a']
>>> class D(C): pass
>>> I = D()
>>> I.a = 1; I.b = 2
>>> I.__dict__
{'b': 2}

# Bullet 2: slots in super but not sub
# Makes instance dict for nonslots
# But slot name still managed in class
```

```

>>> C.__dict__.keys()
dict_keys([... '__slots__', 'a', ...])

>>> class C: __slots__ = ['a']          # Bullet 3: only lowest slot accessible
>>> class D(C): __slots__ = ['a']       # Superclass slot 'a' is pointless

>>> class C: __slots__ = ['a']; a = 99  # Bullet 4: no class-level defaults
ValueError: 'a' in __slots__ conflicts with class variable

>>> class C: __slots__ = ['a']          # Bullet 5: only one nonempty in mixins
>>> class D: __slots__ = ['a']         # Use empty slots or omit in all but one
>>> class E(C, D): pass
TypeError: multiple bases have instance lay-out conflict

```

In other words, besides their program-breaking potential, slots essentially require both *universal* and *careful* deployment to be effective—because slots do not compute values dynamically like properties (coming up in the next section), they are largely pointless unless each class in a tree uses them and is cautious to define only new slot names not defined by other classes. It's an *all-or-nothing* feature—an unfortunate property shared by the super call discussed ahead:

```

>>> class C: __slots__ = ['a']          # Assumes universal use, differing names
>>> class D(C): __slots__ = ['b']
>>> I = D()                             # And may break code and tools you use
>>> I.a = 1; I.b = 2
>>> I.__dict__
AttributeError: 'D' object has no attribute '__dict__'. Did you mean: '__dir__'?
>>> C.__dict__.keys(), D.__dict__.keys()
(dict_keys([... '__slots__', 'a', ...]), dict_keys([... '__slots__', 'b', ...]))

```

Such rules—and others omitted here for space—are part of the reason slots are not widely used and are not generally recommended except in pathological cases where their space reduction is significant. Even then, their potential to complicate or break code should be ample cause to carefully consider the trade-offs. Not only must they be spread almost *neurotically* throughout a framework, but they may also break tools you rely on.

Example impacts of slots: ListTree and mapattr

As a more realistic example of slots' effects, due to the first bullet in the prior section, [Chapter 31](#)'s `ListTree` class ([Example 31-13](#)) does *not fail* when mixed into a class that defines `__slots__`, even though it scans instance namespace dictionaries without verifying their presence. This lister class's own lack of slots is enough to ensure that the instance will still have a `__dict__` and hence not trigger an exception when fetched or indexed.

For example, both of the following *single*-inheritance trees display without error—the second also allows names not in the slots list to be assigned as instances attributes, including any required by the superclass:

```

class C(ListTree): pass
I = C()                                # OK: no __slots__ used
print(I)

class C(ListTree): __slots__ = ['a', 'b'] # OK: superclass produces __dict__
I = C()
I.c = 3
print(I)                               # Displays c at I, a and b at C

```

The following *multiple*-inheritance classes display correctly as well—*any* nonslot class like `ListTree` generates an instance `__dict__` and can thus safely assume its presence. Although it renders subclass slots pointless, this is a positive side effect for tool classes like `ListTree` and its [Chapter 28](#) predecessor:

```
class A: __slots__ = ['a']                # Both OK by bullet 1 above
class B(A, ListTree): pass
print(B())

class A: __slots__ = ['a']
class B(A, ListTree): __slots__ = ['b']    # Displays b at B, a at A
print(B())
```

In general, though, tools may need to catch exceptions when `__dict__` is absent or use a `hasattr` or `getattr` to test or provide defaults if slot usage may preclude an instance namespace dictionary. For instance, [Chapter 31](#)'s `mapattrs.py` module ([Example 31-14](#)) must check for `__dict__` presence explicitly because it is not a class mixed into others, and so cannot assume this attribute. Like `ListTree`, this example also associates slots with their classes.

Run these examples on your own for more info. Slots' impacts may be onerous, but knowledge is your best defense.

What about slots speed?

Finally, while slots primarily optimize memory use, their speed impact is less clear-cut. [Example 32-4](#) codes a simple test script using the `timeit` techniques we studied in [Chapter 21](#). For both the slots and nonslots (instance dictionary) storage models, it makes 1,000 instances, assigns and fetches 4 attributes on each, and repeats 1,000 times—for both models taking the best of 5 runs that each exercise a total of 8M attribute operations.

Example 32-4. slots-test.py

```
import timeit
base = """
Is = []
for i in range(1000):
    I = C()
    I.a = 1; I.b = 2; I.c = 3; I.d = 4
    t = I.a + I.b + I.c + I.d
    Is.append(I)
"""

stmt = """
class C:
    __slots__ = ['a', 'b', 'c', 'd']
""" + base
print('Slots    =>', end=' ')
print(min(timeit.repeat(stmt, number=1000, repeat=5)))

stmt = """
class C:
    pass
""" + base
print('Nonslots=>', end=' ')
print(min(timeit.repeat(stmt, number=1000, repeat=5)))
```

At least for this code, on the macOS test host, and using CPython 3.12, the best times imply that slots are only slightly quicker, though this says little about memory space and is prone to change arbitrarily in the future (PyPy 7.3 struggled on this test with times 10x slower than CPython, presumably due to dynamic class creation, but relatively similar):

```
$ python3 slots-test.py
Slots    => 0.17895982996560633
Nonslots=> 0.18887511501088738
```

For more on slots in general, see the Python standard manual set. Also, watch for the `Private` decorator case study of [Chapter 39](#)—an example that naturally allows for attributes based on both `__slots__` and `__dict__` storage, by using delegation and storage-neutral accessor tools like `getattr`.

Properties: Attribute Accessors

Our next attribute-related topic is *properties*—a mechanism that provides another way for classes to define methods called automatically for access or assignment to instance attributes. This feature is similar to “getters” and “setters” in languages like Java and C#, but in Python is generally best used sparingly as a way to add accessors to attributes *after the fact* as needs evolve and warrant. Where needed, though, properties allow attribute values to be computed dynamically without requiring method calls at the point of access.

Though properties cannot support generic attribute routing goals, at least for specific attributes, they are an alternative to some traditional uses of the `__getattr__` and `__setattr__` overloading methods we first studied in [Chapter 30](#). Properties can have a similar effect to these two methods but, by contrast, incur an extra method call only for accesses to names that require dynamic computation—other nonproperty names are accessed normally with no extra calls. Although `__getattr__` is invoked only for *undefined* names, the `__setattr__` method is instead called for assignment to *every* attribute.

Properties and slots are related, too, but serve different goals. Both implement instance attributes that are not physically stored in instance namespace dictionaries—a sort of “virtual” attribute—and both are based on the notion of class-level attribute *descriptors*. In contrast, slots manage instance storage, while properties intercept access and compute values arbitrarily. Because their underlying descriptor implementation tool is too advanced for us to cover here, properties and descriptors both get full treatment in [Chapter 38](#).

Property basics

As a brief introduction, though, a property is a type of object assigned to a class attribute name. You can generate a property by calling the `property` built-in function, passing in up to three accessor methods—handlers for get, set, and delete operations—as well as an optional docstring for the property. If any argument is passed as `None` or omitted, that operation is not supported.

The resulting property object is typically assigned to a name at the top level of a `class` statement as a class attribute (e.g., `name=property(...)`), and a special `@` decorator syntax you’ll meet later is available to automate this step. When thus assigned, later accesses to the class property name itself as an object attribute (e.g., `obj.name`) are automatically routed to one of the accessor methods passed into the property call.

For example, we’ve seen how the `__getattr__` operator-overloading method allows classes to intercept *undefined* attribute references:

```

>>> class WithOperators:
    def __getattr__(self, name): # On undefined attr
        if name == 'age':
            return 40
        else:
            raise AttributeError(name)

>>> x = WithOperators()
>>> x.age # Runs __getattr__
40
>>> x.name # Runs __getattr__
AttributeError: name

```

Here is the same example, coded with *properties* instead:

```

>>> class WithProperties:
    def getage(self):
        return 40
    age = property(getage) # (get?, set?, del?, docs?), or @

>>> x = WithProperties()
>>> x.age # Runs getage
40
>>> x.name # Normal fetch
AttributeError: 'WithProperties' object has no attribute 'name'

```

For some coding tasks, properties can be less complex and quicker to run than the traditional techniques. For example, when we add attribute *assignment* support, properties become more attractive—there’s less code to type, and no extra method calls are incurred for assignments to attributes we don’t wish to manage or compute dynamically:

```

>>> class WithProperties:
    def getage(self):
        print('get age')
        return 40
    def setage(self, value):
        print('set age:', value)
        self._age = value
    age = property(getage, setage)

>>> x = WithProperties()
>>> x.age # Runs getage
get age
40
>>> x.age = 42 # Runs setage
set age: 42
>>> x._age # Normal fetch: no getage call
42
>>> x.job = 'hacker' # Normal assign: no setage call
>>> x.job # Normal fetch: no getage call
'hacker'

```

The equivalent class based on operator overloading incurs extra method calls for assignments to attributes not being managed and needs to route attribute assignments through the attribute dictionary to avoid loops (or to the object superclass’s `__setattr__` to better support “virtual” attributes such as slots and properties coded in other classes):

```

>>> class WithOperators:
    def __getattr__(self, name):          # On undefined attr
        if name == 'age':
            print('get age')
            return 40
        else:
            raise AttributeError(name)
    def __setattr__(self, name, value):   # On all assignments
        print('set:', name, value)
        if name == 'age':
            self.__dict__['age'] = value  # Or object.__setattr__(self, ...)
        else:
            self.__dict__[name] = value

>>> x = WithOperators()
>>> x.age                                # Runs __getattr__
get age
40
>>> x.age = 41                           # Runs __setattr__
set: age 41
>>> x._age                               # Defined: no __getattr__ call
41
>>> x.job = 'coder'                     # Runs __setattr__ again
set: job coder
>>> x.job                               # Defined: no __getattr__ call
'coder'

```

Properties seem like a win for this simple example. However, some applications of `__getattr__` and `__setattr__` still require more dynamic or generic interfaces than properties directly provide.

For example, the set of attributes to be managed might be unknown when a class is coded and may not even exist in a tangible form (e.g., when *delegating* arbitrary attribute references to a wrapped and embedded object generically). In such contexts, a generic attribute handler like `__getattr__` with a passed-in attribute name may be preferable. Because such generic handlers can also support simpler cases, properties may be a redundant extension—albeit one that may avoid extra calls on assignments and one that some programmers may prefer when applicable.

For more details on both options, tune in to [Chapter 38](#). As you'll see there, it's also possible to code properties using the `@` symbol *function decorator* syntax—a topic introduced in brief later in this chapter and an equivalent and automatic alternative to manual assignment in the class scope:

```

class WithProperties:
    @property
    def age(self):                        # Coding properties with decorators: ahead
        ...                               # On instance.age
    @age.setter
    def age(self, value):                 # On instance.age = value
        ...

```

To make sense of this decorator syntax, though, we must move ahead.

`__getattribute__` and Descriptors: Attribute Implementations

To complete our attribute-tools collection, the `__getattribute__` operator-overloading method intercepts *all* attribute references, not just undefined references. This makes it more potent than its `__getattr__` cousin we used in the prior section, but also trickier to use—it's prone to loops much like `__setattr__`, but in different ways.

For more specialized attribute interception goals, in addition to properties and operator-overloading methods, Python provides attribute *descriptors*—classes with `__get__` and `__set__` methods, assigned to class attributes and inherited by instances, that intercept read and write accesses to specific attributes. As a preview, here’s one of the simplest descriptors you’re likely to encounter:

```
>>> class AgeDesc:
    def __get__(self, instance, owner): return 40
    def __set__(self, instance, value): instance._age = value

>>> class WithDescriptors:
    age = AgeDesc()           # Assign descriptor instance

>>> x = WithDescriptors()
>>> x.age                     # Runs AgeDesc.__get__
40
>>> x.age = 42               # Runs AgeDesc.__set__
>>> x._age                   # Normal fetch: no AgeDesc call
42
```

Descriptors have access to state-information attributes in instances of themselves as well as their client class and are, in a sense, a more general form of properties. In fact, *properties* are a simplified way to define a specific type of descriptor—one that runs functions on access. Descriptors are also used to implement the *slots* feature we met earlier, among other Python tools, and are afforded special cases in attribute *inheritance* alluded to earlier in this chapter.

Because `__getattr__` and descriptors are too substantial to present here, we’ll defer the rest of their coverage, as well as much more on properties, to [Chapter 38](#). We’ll also employ them in examples in [Chapter 39](#) and study how they factor into inheritance in [Chapter 40](#). Here, the topics tour is moving on.

Static and Class Methods

Beyond the usual methods we’ve been using so far, classes can define two kinds of methods called without an instance: *static* methods work roughly like simple instance-less functions inside a class no matter how they’re called, and *class* methods are passed a class instead of an instance. Both are similar to tools in other languages (e.g., C++ static methods). The prior chapter’s bound method coverage noted these briefly, but we’ll finish the story here.

To enable these special method modes, you call built-in functions named `staticmethod` and `classmethod` within the class or invoke them with the special `@name` decoration syntax you’ll meet later in this chapter. The `classmethod` call is required to enable its mode; `staticmethod` is not required for instance-less methods called only through a class name but is required if such methods are called through instances.

Why the Special Methods?

As we’ve seen, a class’s method is normally passed an instance object in its first argument to serve as the implied subject of the method call—that’s the “object” in “object-oriented programming.” Though much less common, there are two formal ways to temper this model. Before we get to the syntax, let’s clarify why this might matter to you.

Sometimes, programs need to process data associated with *classes* instead of instances. Consider keeping track of the number of instances created from a class or maintaining a list of all of a class’s instances that are currently in use. This type of information and its processing are associated with the class rather

than its instances. That is, the information is usually stored on the class itself and processed apart from any instance.

For such tasks, simple functions coded outside a class might suffice—because they can access class attributes through the class name, they have access to class data, and never require access to an instance. However, to better associate such code with a class and to allow such processing to be customized with inheritance as usual, it would be better to code these types of functions *inside* the class itself. To make this work, we need methods in a class that are not passed, and do not expect, a `self` instance argument.

Per the prior chapter, methods accessed through the class are *plain functions* that meet some of this need but fail if accessed through an instance: the resulting *bound method* passes an instance in calls, even if the plain function doesn't expect one. To address, Python provides *static methods*—plain functions that are nested in a class and never expect nor receive an automatic `self` argument, regardless of how they are called. They're optional for methods only ever accessed through classes but needed for access through instances.

Although less commonly used, Python also supports *class methods*—methods of a class that are passed a class object in their first argument instead of an instance, regardless of whether they are called through an instance or a class. Such methods can access class data through their class argument—what we've called `self` thus far—even if called through an instance. Normal methods, sometimes called *instance methods*, still receive a subject instance when called; static and class methods do not.

Plain-Function Methods

To demo the preceding ideas, let's suppose that we want to use class attributes to count how many instances are generated from a class. [Example 32-5](#), *hack1.py*, makes a first attempt—its class has a counter stored as a class attribute, a constructor that bumps up the counter by one each time a new instance is created, and a method that displays the counter's value. Remember, class attributes are stored just once on a class and shared by all instances; storing the counter this way ensures that it effectively spans all instances.

Example 32-5. hack1.py

```
class Hack:
    numInstances = 0
    def __init__(self):
        Hack.numInstances += 1
    def printNumInstances():
        print('Number of instances created:', Hack.numInstances)
```

The `printNumInstances` method is designed to process class data, not instance data—it's about *all* the instances, not any one in particular. Because of that, we want to be able to call it without having to pass an instance. Indeed, we don't want to *make* an instance to fetch the number of instances because this would *change* the number of instances we're trying to fetch! In other words, we want a `self`-less “static” method.

Whether this code's `printNumInstances` works or not, though, depends on which way you call the method—through the class or through an instance. Calls to `self`-less methods made through classes work because they produce plain functions, but calls from instances produce bound methods and fail:


```

$ python3
>>> from hack1 import Hack
>>> a, b, c = Hack(), Hack(), Hack()      # Make three instances

>>> Hack.printNumInstances()              # Okay to call from instance - only!
Number of instances created: 3
>>> a.printNumInstances()
TypeError: Hack.printNumInstances() takes 0 positional arguments but 1 was given

```

Calls to instance-less methods like `printNumInstances` made through the *class* work, but calls made through an *instance* fail because an instance is automatically passed to a method that does not have an argument to receive it. If you're able to stick with calling *self*-less methods through classes only, you already have a static method. However, to allow *self*-less methods to be called through instances, you need to either adopt other designs or mark such methods as special. Let's look at both options in turn.

Static Method Alternatives

Short of marking a *self*-less method as special, you can sometimes achieve similar results with different coding structures. For example, if you just want to call functions that access class members without an instance, perhaps the simplest idea is to use normal functions outside the class, not class methods. This way, an instance isn't expected in the call. The mutation in [Example 32-6](#) illustrates.

Example 32-6. *hack2.py*

```

def printNumInstances():
    print('Number of instances created:', Hack.numInstances)

class Hack:
    numInstances = 0
    def __init__(self):
        Hack.numInstances += 1

```

Because the class name is accessible to the simple function as a global variable, this works fine. Also, note that the name of the function becomes global, but only to this single module; it will not clash with names in other files:

```

>>> import hack2 as hack
>>> a = hack.Hack()
>>> b = hack.Hack()
>>> c = hack.Hack()
>>> hack.printNumInstances()              # But function may be too far removed
Number of instances created: 3            # And cannot be changed via inheritance
>>> hack.Hack.numInstances
3

```

Prior to static methods in Python, this structure was the general prescription. Because Python already provides modules as a namespace-partitioning tool, one could argue that there's not typically any need to package functions in classes unless they implement object behavior. Simple functions within modules like the one here do much of what instance-less class methods could and are already associated with the class because they live in the same module.

This approach, though, may be subpar. For one thing, it adds to this file's scope an extra name that is used only for processing a single class. For another, the function is not directly associated with the class by structure; in fact, its `def` could be hundreds of lines away. Worse, simple functions like this cannot be customized by inheritance since they live outside a class's namespace: subclasses cannot directly replace or extend such a function by redefining it.

We might also try to make this example work by simply using a normal method and always calling it through an instance, as usual. Unfortunately, such an approach is completely unworkable if we don't have an instance available, and making an instance changes the class data, as noted earlier. A better solution would be to somehow mark a method inside a class as never requiring an instance. The next section shows how.

Using Static and Class Methods

To designate a *self*-less method that may be called through *either* the class or its instances, classes can simply call the built-in functions `staticmethod` and `classmethod`. Both mark a function object as special—requiring no instance for the former and requiring a class argument for the latter. [Example 32-7](#) shows how.

Example 32-7. *allmethods.py*

```
class Methods:
    def imeth(self, x):          # Instance method: passed a self
                                # Always expects a self instance

    def smeth(x):                # Static method: no instance passed
                                # Also a plain function from the class

    def cmeth(cls, x):           # Class method: gets class, not instance
                                # Always expects a class, not instance

    smeth = staticmethod(smeth)  # Make smeth a static method (or use @: ahead)
    cmeth = classmethod(cmeth)  # Make cmeth a class method (or use @: ahead)
```

Notice how the last two assignments in this code simply *reassign* (a.k.a. *rebind*) the method names `smeth` and `cmeth`. Attributes are created and changed by any assignment in a class statement, so these final assignments simply overwrite the assignments made earlier by the `defs`. As you'll see in a few moments, the special `@` decorator syntax works here as an alternative to this just as it does for properties—but makes little sense unless you first understand the assignment form here that it automates.

Technically, Python supports three kinds of class-related methods with differing argument protocols:

- *Instance methods*, passed a `self` instance object (the default)
- *Static methods*, passed no extra instance object (via `staticmethod`)
- *Class methods*, passed a class object (via `classmethod`, and inherent in metaclasses)

Moreover, simple functions in a class also serve the role of static methods without requiring any extra protocol when called through a class object only. The *allmethods.py* module illustrates all three method types, so let's expand on these in turn.

Instance methods are the normal and default case that we've used in this book so far. An instance method must always be called with an instance object. When you call it through an *instance*, Python passes the instance to the first (leftmost) argument automatically; when you call it through a *class*, you must pass along the instance manually:

```
>>> from allmethods import Methods    # Normal instance methods
>>> obj = Methods()                  # Callable through instance or class
>>> obj.imeth(1)                     # Becomes imeth(obj, 1)
[<allmethods.Methods object at 0x1015a79b0>, 1]
```

```
>>> Methods.imeth(obj, 2)
[<allmethods.Methods object at 0x1015a79b0>, 2]
```

Static methods, by contrast, are called without an instance argument. Unlike simple functions outside a class, their names are local to the scopes of the classes in which they are defined, and they may be looked up by inheritance. Instance-less functions can be called through a class normally, but using the `staticmethod` built-in allows such methods to also be called through an instance. That is, the first of the following works without the `staticmethod` in the class but the second does not:

```
>>> Methods.smeth(3)                # Static method: call through class
[3]                                # No instance passed or expected
>>> obj.smeth(4)                    # Static method: call through instance
[4]                                # Instance not passed - requires staticmethod
```

Class methods are similar, but Python automatically passes the class (not an instance) to a class method's first (leftmost) argument, whether it is called through a class or an instance:

```
>>> Methods.cmeth(5)                # Class method: call through class
[<class 'allmethods.Methods'>, 5]   # Becomes cmeth(Methods, 5)
>>> obj.cmeth(6)                    # Class method: call through instance
[<class 'allmethods.Methods'>, 6]   # Becomes cmeth(Methods, 6)
```

In [Chapter 40](#), you'll also find that *metaclass methods*—an advanced and technically distinct method type used in the secondary class trees of types—behave similarly to the explicitly declared class methods we're exploring here.

Counting Instances with Static Methods

Now, given these built-ins, [Example 32-8](#) codes the static method equivalent of this section's instance-counting example—it marks the method as special, so it will never be passed an instance automatically.

Example 32-8. hack_static.py

```
class Hack:
    numInstances = 0                # Use static method for class data
    def __init__(self):
        Hack.numInstances += 1
    def printNumInstances():
        print('Number of instances:', Hack.numInstances)
    printNumInstances = staticmethod(printNumInstances)
```

Using the static method built-in, our code now allows the `self`-less method to be called through the class or any instance of it:

```
>>> from hack_static import Hack
>>> a, b, c = Hack(), Hack(), Hack()
>>> Hack.printNumInstances()        # Call as simple function
Number of instances: 3
>>> a.printNumInstances()          # Instance argument not passed
Number of instances: 3
```

Compared to simply moving `printNumInstances` outside the class, as prescribed earlier, this version requires an extra `staticmethod` call (or an `@` line you'll meet ahead). However, it also localizes the function name in the class scope (so it won't clash with other names in the module); moves the function code closer to where it is used (inside the `class` statement); and allows subclasses to *customize* the static method with inheritance—a more convenient and powerful approach than importing functions from

the files in which superclasses are coded. The following subclass illustrates (this continues the prior session, so the count is already 3 at the start):

```
>>> class Sub(Hack):
    def printNumInstances():          # Override a static method
        print('Extra stuff...')     # But call back to original
        Hack.printNumInstances()
    printNumInstances = staticmethod(printNumInstances)

>>> a, b = Sub(), Sub()
>>> a.printNumInstances()           # Call from subclass instance
Extra stuff...
Number of instances: 5
>>> Sub.printNumInstances()        # Call from subclass itself
Extra stuff...
Number of instances: 5
>>> Hack.printNumInstances()       # Call original version
Number of instances: 5
```

Moreover, classes can inherit the static method without redefining it—it is run without an instance, regardless of where it is defined in a class tree:

```
>>> class Other(Hack): pass        # Inherit static method verbatim

>>> c = Other()
>>> c.printNumInstances()
Number of instances: 6
```

Notice how this also bumps up the *superclass*'s instance counter because its constructor is inherited and run—a behavior that begins to encroach on the next section's subject.

Counting Instances with Class Methods

Interestingly, a *class method* can do similar work here—[Example 32-9](#) has the same behavior as the static method version listed earlier, but it uses a class method that receives the instance's class in its first argument. Rather than hardcoding the class name, the class method uses the automatically passed class object generically.

Example 32-9. *hack_class.py*

```
class Hack:
    numInstances = 0                # Use class method instead of static
    def __init__(self):
        Hack.numInstances += 1
    def printNumInstances(cls):
        print('Number of instances:', cls.numInstances)
    printNumInstances = classmethod(printNumInstances)
```

This class is used in the same way as the prior versions, but its `printNumInstances` method receives the `Hack` class, not the instance, when called from either the class or an instance:

```
>>> from hack_class import Hack
>>> a, b = Hack(), Hack()
>>> a.printNumInstances()          # Passes class to first argument
Number of instances: 2
>>> Hack.printNumInstances()      # Also passes class to first argument
Number of instances: 2
```

When using class methods, though, keep in mind that they receive the most specific (i.e., *lowest*) class of the call's subject. This has some subtle implications when trying to update class data through the passed-in class. To demo, [Example 32-10](#) subclasses to customize the same way we did for static methods in the prior section, and augments `Hack.printNumInstances` to also trace its `cls` argument.

Example 32-10. *hack_class2.py*

```
class Hack:
    numInstances = 0                                # Trace class passed in
    def __init__(self):
        Hack.numInstances += 1
    def printNumInstances(cls):
        print('Number of instances:', cls.numInstances, cls)
    printNumInstances = classmethod(printNumInstances)

class Sub(Hack):
    def printNumInstances(cls):                      # Override a class method
        print('Extra stuff...', cls)                # But call back to original
        Hack.printNumInstances()
    printNumInstances = classmethod(printNumInstances)

class Other(Hack): pass                            # Inherit class method verbatim
```

Running this in a REPL reveals that the lowest class is passed in whenever a class method is run—even for subclasses that have no class methods of their own:

```
>>> from hack_class2 import Hack, Sub, Other
>>> x = Sub()
>>> y = Hack()

>>> x.printNumInstances()                          # Call from subclass instance
Extra stuff... <class 'hack_class2.Sub'>
Number of instances: 2 <class 'hack_class2.Hack'>

>>> Sub.printNumInstances()                        # Call from subclass itself
Extra stuff... <class 'hack_class2.Sub'>
Number of instances: 2 <class 'hack_class2.Hack'>

>>> y.printNumInstances()                          # Call from superclass instance
Number of instances: 2 <class 'hack_class2.Hack'>
```

In the first call here, a class method call is made through an instance of the `Sub` subclass, and Python passes the lowest class, `Sub`, to the class method. All is well in this case—since `Sub`'s redefinition of the method calls the `Hack` superclass's version explicitly, the superclass method in `Hack` receives its own class in its first argument. But watch what happens for an object that inherits the class method verbatim:

```
>>> z = Other()                                    # Call from lower sub's instance
>>> z.printNumInstances()
Number of instances: 3 <class 'hack_class2.Other'>
```

This last call here passes `Other` to `Hack`'s class method. This works in this example because *fetching* the counter finds it in `Hack` by class inheritance. If this method tried to *assign* to the passed class's data, though, it would update `Other`, not `Hack`! In this specific case, `Hack` is probably better off hardcoding its own class name to update its data if it means to count instances of all its subclasses, too, rather than relying on the passed-in class argument.

Counting instances per class with class methods

In fact, because class methods always receive the *lowest* class in an instance's tree:

- *Static* methods and explicit class names may be a better solution for processing data local to a class.
- *Class* methods may be better suited to processing data that may differ for each class in a hierarchy.

Code that needs to manage *per-class* instance counters, for example, might be best off leveraging class methods. To illustrate, the top-level superclass in [Example 32-11](#) uses a class method to manage state information that varies for and is stored on each class in the tree—similar in spirit to the way instance methods manage state information that varies per class instance.

Example 32-11. *hack_class3.py*

```
class Hack:
    numInstances = 0
    def count(cls):
        cls.numInstances += 1
    def __init__(self):
        self.count()
    count = classmethod(count)

class Sub(Hack):
    numInstances = 0
    def __init__(self):
        Hack.__init__(self)

class Other(Hack):
    numInstances = 0
```

When run, the Hack class keeps track of each of its subclasses' instances, using a counter on each subclass:

```
>>> from hack_class3 import Hack, Sub, Other
>>> x = Hack()
>>> y1, y2 = Sub(), Sub()
>>> z1, z2, z3 = Other(), Other(), Other()
>>> x.numInstances, y1.numInstances, z1.numInstances
(1, 2, 3)
>>> Hack.numInstances, Sub.numInstances, Other.numInstances
(1, 2, 3)
```

Static and class methods have additional advanced roles, which we will skip here; see other resources for more use cases. In later Python versions, though, the static and class method designations became even simpler with the advent of *function decoration* syntax—a way to apply one function to another that has roles well beyond the static method use case that was one of its initial motivations. This syntax also allows us to augment *classes*—to initialize data like the `numInstances` counter in the last example, for instance. The next section explains how.



The methods finale: For a postscript on Python's method types, be sure to watch for coverage of metaclass methods in [Chapter 40](#)—because these are designed to process a *class* that is an instance of a metaclass, they turn out to be very similar to the class methods defined here but require no `classmethod` declaration, and apply only to the shadowy metaclass realm previewed next.

Decorators and Metaclasses

Because the `staticmethod` and `classmethod` call technique described in the prior section initially seemed obscure to some observers, a device was eventually added to make the operation simpler. Python *decorators*—similar to the notion and syntax of annotations in Java—both address this specific need and provide a general tool for adding logic that manages functions and classes or later calls to them.

This is called a “decoration,” but in more concrete terms is really just a way to run extra processing steps at function and class definition time with explicit syntax. It comes in two flavors:

- *Function decorators*: The initial entry, augment function definitions at `def` statements. They specify operation modes for both simple functions and classes’ methods by wrapping them in an extra layer of logic implemented as another function. That function is often called a *metafunction*, though this is just terminology.
- *Class decorators*: A later extension, augment class definitions at `class` statements. They wrap classes in a similar way, adding support for management of whole objects and their interfaces instead of a single function.

We met decorators very briefly in [Chapter 19](#) in relation to simple functions, but they are more general than earlier implied: they can also be used for class methods and classes and can add nearly arbitrary logic to functions and classes that go well beyond that the static- and class-method roles used as a segue here.

For instance, *function decorators* may be used to augment functions with code that logs calls made to them, checks argument types during debugging, times calls, and so on, and can be used to manage either functions themselves or later calls to them. In the latter mode, function decorators are similar to the *delegation* design pattern we explored in [Chapter 31](#), but they are designed to augment a specific function or method call, not an entire object interface.

Python provides a few built-in function decorators for operations, such as marking static and class methods and defining properties (as sketched earlier, the `property` built-in works as a decorator automatically), but programmers can also code arbitrary decorators of their own. Although they are not strictly tied to classes, user-defined function decorators are often coded as classes to save the original functions for later dispatch, along with other data as state information.

This proved such a useful hook that it was eventually extended—*class decorators* bring augmentation to classes, too, and are more directly tied to the class model. Like their function cohorts, class decorators may manage classes themselves or later instance-creation calls and often employ *delegation* of entire interfaces in the latter mode. As you’ll find, their roles also often overlap with *metaclasses* but are a more lightweight way to achieve some goals.

Function Decorator Basics

Syntactically, a function decorator is a sort of runtime declaration about the function that follows it. A function decorator is coded on a line by itself just before the `def` statement that defines a function or method. It consists of the `@` symbol, followed by a *metafunction*—a plain function (or other callable object) of one argument, which is passed and manages another function. The code following the `@` is usually the name of a metafunction with an optional arguments list, but as of Python 3.9, it can be any expression returning a one-argument function. Listing multiple decorators on consecutive lines allows them to nest, as you’ll see later in this book.

For example, the prior section's methods may be coded with decorator syntax like this:

```
class C:
    @staticmethod
    def meth():
        ...
        # Function decoration syntax
```

Internally, this syntax has the same effect as the following—passing the function through the decorator and assigning the result back to the original name:

```
class C:
    def meth():
        ...
    meth = staticmethod(meth)
        # Name rebinding equivalent
```

Decoration *rebinds* the method name to the decorator's result. The net effect is that calling the method function's name later actually triggers the result of its `staticmethod` decorator first. Because a decorator can return any sort of object, this allows the decorator to insert a layer of logic to be run on every later call. The decorator function is free to return either the original function itself or a new *proxy* object that saves the original function passed to the decorator to be invoked indirectly after the extra logic layer runs.

With this addition, [Example 32-12](#) is a better way to code our static method code of [Example 32-8](#).

Example 32-12. hack_static_deco.py

```
class Hack:
    numInstances = 0
    def __init__(self):
        Hack.numInstances += 1

    @staticmethod
    def printNumInstances():
        print('Number of instances:', Hack.numInstances)
```

Here is this example in action as before:

```
>>> from hack_static_deco import Hack
>>> a, b, c = Hack(), Hack(), Hack()
>>> Hack.printNumInstances()
Number of instances: 3
>>> a.printNumInstances()
Number of instances: 3
# Calls from classes and instances work
```

Because they also accept and return functions, the `classmethod` and `property` built-in functions may be used as decorators in the same way—as in [Example 32-13](#), which demos all three built-in decorators previewed earlier.

Example 32-13. alldecorators.py

```
class Methods:
    def imeth(self, x):
        print([self, x])
        # Normal instance method: passed a self

    @staticmethod
    def smeth(x):
        print([x])
        # Static: no instance passed

    @classmethod
```



```

def cmeth(cls, x):                # Class: gets class, not instance
    print([cls, x])

@property                        # Property: computed on fetch
def name(self):
    return 'Pat ' + self.__class__.__name__

```

Running this live in a REPL proves the point:

```

>>> from alldecorators import Methods
>>> obj = Methods()
>>> obj.imeth(1)
[<alldecorators.Methods object at 0x10d2839b0>, 1]
>>> obj.smeth(2)
[2]
>>> obj.cmeth(3)
[<class 'alldecorators.Methods'>, 3]
>>> obj.name
'Pat Methods'

```

Bear in mind that `staticmethod` and its kin here are still built-in functions; they may be used in decoration syntax just because they take a function as an argument and return a callable to which the original function name can be rebound. In fact, any such function can be used in this way—even user-defined functions we code ourselves, as the next section explains.

A First Look at User-Defined Function Decorators

Although Python provides a handful of built-in functions that can be used as decorators, we can also write custom decorators of our own. Because of their wide utility, we’re going to devote an entire chapter to coding decorators in the final part of this book. As a quick example, though, let’s look at a simple user-defined decorator at work.

Recall from [Chapter 30](#) that the `__call__` operator-overloading method implements a function-call interface for class instances. [Example 32-14](#) uses this to code a call *proxy* class that saves the decorated function in the instance and catches calls to the original name. Because this is a class, it also has state information—a counter of calls made.

Example 32-14. tracer1.py

```

class tracer:
    def __init__(self, func):        # Remember original, init counter
        self.calls = 0
        self.func = func
    def __call__(self, *args):      # On later calls: add logic, run original
        self.calls += 1
        print(f'call {self.calls} to {self.func.__name__}')
        return self.func(*args)

@tracer
def hack(a, b, c):                # Same as hack = tracer(hack)
    return a + b + c              # Wrap hack in a decorator object

if __name__ == '__main__':
    print(hack(1, 2, 3))           # Really calls the tracer wrapper object
    print(hack('a', 'b', 'c'))    # Invokes __call__ in class

```

Because the hack function is run through the tracer decorator, when the original hack name is called, it actually triggers the `__call__` method in the class. This method counts and logs the call and then dispatches it to the original wrapped function. Note how the **name* argument syntax is used to pack and unpack the passed-in arguments; because of this, this decorator can be used to wrap any function with any number of positional arguments.

The net effect, again, is to add a layer of logic to the original hack function. When run, the first output line comes from the tracer class, and the second gives the return value of the hack function itself:

```
$ python3 tracer1.py
call 1 to hack
6
call 2 to hack
abc
```

Trace through this example's code for more insight. As it is, this decorator works for any function that takes positional arguments, but it does not handle *keyword* arguments and cannot decorate class-level *method* functions (in short, for methods, its `__call__` would be passed a tracer instance only). As you'll learn in [Part VIII](#), there are a variety of ways to code function decorators, including nested `def` statements, and some of the alternatives are better suited to methods than the version shown here.

For example, by using *nested functions* with enclosing scopes for state instead of callable class instances with attributes, function decorators often become more broadly applicable to class-level *methods* too. We'll postpone the full details on this, but [Example 32-15](#) provides a brief look at this *closure*-based coding model; it uses function attributes for counter state for portability but could also leverage variables and `nonlocal` instead.

Example 32-15. *tracer2.py*

```
def tracer(func):
    # Remember original
    def oncall(*args):
        # On later calls
        oncall.calls += 1
        print(f'call {oncall.calls} to {func.__name__}')
        return func(*args)
    oncall.calls = 0
    return oncall

class C:
    @tracer
    def hack(self, a, b, c): return a + b + c

if __name__ == '__main__':
    x = C()
    print(x.hack(1, 2, 3))
    print(x.hack('a', 'b', 'c'))
```

The example's output is the same as its predecessor but reflects a decorated class method; more on this later.

A First Look at Class Decorators and Metaclasses

Python later generalized decorators, allowing them to be applied to classes as well as functions. In short, *class decorators* are similar to function decorators, but they are run at the end of a `class` statement to rebind a class name to a callable. As such, they can be used to either manage classes just after they are

created or insert a layer of wrapper logic to manage instances when they are later created. Symbolically, the code structure:

```
def decorator(aClass): ...

@decorator                                # Class decoration syntax
class C: ...
```

is mapped to the following equivalent:

```
def decorator(aClass): ...

class C: ...                                # Name rebinding equivalent
C = decorator(C)
```

The class decorator is free to augment the class itself or return a *proxy* object that intercepts later instance construction calls. For example, in the code of “Counting instances per class with class methods” on page 844, we could use this hook to automatically augment the classes with instance counters and any other data required:

```
def count(aClass):
    aClass.numInstances = 0
    return aClass                                # Return class itself instead of a wrapper

@count
class Hack: ...                                # Same as Hack = count(Hack)

@count
class Sub(Hack): ...                            # numInstances = 0 not needed here
```

In fact, as coded, this decorator can be applied to classes or functions—it happily returns the object being defined in either context after initializing the object’s attribute:

```
@count
def hack(): pass                                # Like hack = count(hack)

@count
class Hack: pass                                # Like Hack = count(Hack)

hack.numInstances                                # Both are set to zero
Hack.numInstances
```

Though this decorator manages a function or class itself, as detailed later in this book, class decorators can also manage an object’s entire *interface* by intercepting construction calls and wrapping the new instance object in a *proxy* that deploys attribute accessor tools to intercept later requests—a multilevel coding technique we’ll use to implement class attribute privacy in Chapter 39. Here’s a preview of the model:

```
def decorator(cls):                                # On @ decoration
    class Proxy:
        def __init__(self, *args):                # On instance creation: make a cls
            self.wrapped = cls(*args)
        def __getattr__(self, name):                # On attribute fetch: extra ops here
            return getattr(self.wrapped, name)
    return Proxy

@decorator
class C: ...                                # Like C = decorator(C)
X = C()                                    # Makes a Proxy that wraps a C, and catches later X.attr
```

Finally, *metaclasses*, mentioned briefly earlier in this chapter, are a similarly advanced class-based tool whose roles often intersect with those of class decorators. They provide an alternate model, which routes the creation of a class object to a subclass of the top-level type class (normally), at the conclusion of a `class` statement:

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):
        ...extra logic + class creation via type call...

class C(metaclass=Meta):
    ...my creation routed to Meta...           # Like C = Meta('C', (), {...})
```

Python calls a class's metaclass to create the new class object, passing in the data defined during the `class` statement's run; if omitted, the `metaclass` simply defaults to the `type` class we explored earlier. Abstractly speaking, here's what happens at the end of `class` statements having explicit metaclasses like the preceding:

```
classname = Meta(classname, superclasses, attributedict)
```

To manage the creation or initialization of a new class object, a metaclass generally redefines the `__new__` or `__init__` method of the `type` class that intercepts this call by default. The net effect, as with class decorators, is to define code to be run automatically at class creation time. Here, this binds the class name to the result of a call to a user-defined metaclass. In fact, a metaclass need not be a class at all—a possibility we'll explore later that blurs some of the distinction between this tool and decorators and even qualifies the two as functionally equivalent in some roles.

Both schemes, class decorators and metaclasses, are free to augment a class or return an arbitrary object to replace it—a hook with almost limitless class-based customization possibilities. As you'll learn later, metaclasses may also define *methods* that process their instance classes rather than normal instances of them—a technique that's similar (if not redundant) to class methods and might be emulated by methods and data in class decorator proxies, or even a class decorator that returns a metaclass instance.

Such mind-bending concepts, however, require [Chapter 40](#)'s conceptual groundwork (and quite possibly sedation).

For More Details

Naturally, there's more to the decorator and metaclass stories than shown here. Although they are a general mechanism whose usage may be required by some packages, coding *new* user-defined decorators and metaclasses is an advanced topic of interest primarily to tool writers, not application programmers. Because of this, this book defers additional coverage until its final and optional part:

- [Chapter 38](#) shows how to code properties using function decorator syntax in more depth.
- [Chapter 39](#) focuses on decorators, and includes more comprehensive examples.
- [Chapter 40](#) covers metaclasses, and more on the class and instance management story.

Although these chapters cover advanced topics, they'll also provide us with a chance to see Python at work in substantial examples. For now, let's move on to our final class-related topic.

The super Function

To close out this chapter, we turn to `super`: a built-in function that can be used to both reference super-class attributes implicitly without naming a superclass explicitly and route method calls coherently in multiple-inheritance trees.

So far, `super` has been called out in the sidebar “[The super Alternative](#)” on page 691, as well as multiple notes along the way, but has not appeared in code. This was by design: `super` comes with substantial complexities and downsides that make it difficult to recommend to learners. In short, it’s an all-or-nothing tool with several arduous coding requirements and relies on special-case and wildly implicit semantics that run counter to Python norms.

The `super` call also tends to be abused for “Java-fication” of Python code. Newcomers with backgrounds in Java often rush to use Python’s `super` simply because of its similarity to a Java tool but are unaware of its much more subtle implications in Python’s multiple inheritance—until adding superclasses breaks their programs.

Nevertheless, this call has grown pervasive in Python code and merits further elaboration, especially for those opting to use it naively. Hence, this section both covers `super` usage and notes its pitfalls along the way. Ultimately, though, the merit of this call, like everything else presented in this chapter and book, is ultimately yours to decide.

The super Basics

First off, let’s review the explicit alternative that’s more closely in step with Python’s own idioms. As we’ve seen in this book so far, it’s always possible to reference a superclass’s attributes—whether method or data—by naming their desired source class explicitly:

```
>>> class C:
    def act(self):
        print('hack')

>>> class D(C):
    def act(self):
        C.act(self)          # Name superclass explicitly, pass self
        print('code')

>>> I = D()
>>> I.act()
hack
code
```

In *single-inheritance* trees like this one, the `super` alternative seems relatively straightforward at first glance: its most common form in the following automatically selects the calling class’s superclass generically and implicitly when an attribute is later fetched. An explicit call like `class.method(self)`, for example, becomes an implicit `super().method()`, as in our example:

```

>>> class C:
    def act(self):
        print('hack')

>>> class D(C):
    def act(self):
        super().act()      # Reference superclass implicitly, omit self
        print('code')

>>> I = D()
>>> I.act()
hack
code

```

This works as advertised and may minimize work—you don’t need to list `self` in the call, don’t need to update the call if D’s superclass changes in the future, and don’t need to code long superclass names or package-import paths.

The super Details

If you study the preceding code closely, though, you’ll realize that there’s something odd going on here. The `super` call somehow knows about the class, its superclass, and the `self` instance, even though none are present in its call. The backstory involves MROs, a proxy, and an algorithm that are required reading for super aspirants of all kinds.

A “magic” proxy

To understand how `super` works, you first need to be fluent in the *MRO algorithm* covered in “[Multiple Inheritance and the MRO](#)” on page 796—and you should review that now if you gave it a pass. The MRO is both a firm prerequisite and nested component of `super`. Given the complexity and artificial nature of the MRO, some may rule this a first strike against `super`.

Once you’ve mastered the MRO, the simplest description of `super` is this: when used in a class method, `super` returns a *proxy* object that will locate an attribute in a class *following* that of the containing class in the MRO of the `self` instance’s class. The net effect finds an attribute in a superclass or other relative of the class containing the call.

This works as expected in single-inheritance trees because the superclass naturally follows the containing class on `self`’s MRO. Really, though, this relies on deep magic. Apart from the MRO itself, `super` works by inspecting:

- The runtime *call stack* info for the calling method’s arguments
- The `__class__` variable internally added to the `__closure__` of methods that call `super`

The combo automatically locates both the `self` argument and the class containing the `super` call and then pairs the two in a special *proxy* object that routes later attribute fetches to a superclass’s version of a name.

In fact, the common no-argument `super` form is equivalent to manually passing in the class containing the `super` call, along with the `self` instance. That is, within a class’s method function, the following forms work the same, though the second can be used outside a method, too, and its first argument can be `__class__` inside a method:

```
super()
super(class-containing-the-super-call, method-self-argument)
```

Both forms can be used in your code, and manual arguments may be handy in some roles. Because the second may be *harder* to code and maintain than explicit class-name references, though, Python roots out the class and instance for you behind the scenes. Here's the equivalent manual version in our example:

```
class D(C):
    def act(self):
        super(D, self).act()    # Works the same as super().act()
        print('code')          # And D is available as __class__
```

If that all sounds complicated and strange, it's because it is. Due to its unusual semantics, the no-argument `super` call form doesn't work at all outside the context of a class's method:

```
>>> super                # A "magic" proxy object that routes later calls
<class 'super'>
>>> super()              # This form has no meaning outside a method
RuntimeError: super(): no arguments
```

And where it does work, its implicit pairing of class and instance is nowhere to be found in your code:

```
>>> class E(C):
    def method(self):
        proxy = super()    # self is implicit in super - only!
        return proxy

>>> prx = E().method()    # The normally hidden proxy object
>>> prx
<super: <class 'E'>, <E object>>

>>> prx.act()            # Find act on MRO past hidden E, bind with hidden self, call (!)
Hack
```

To be sure, this call's semantics resemble nothing else in Python—it's neither a bound nor nonbound method and fills in a class and `self` even though you omit both in the call. This deviates from Python's explicit `self` policy, which holds true everywhere else. As we've seen, class methods list and use `self` explicitly to make instance references apparent. Operator overloading, including constructors, implies a `self`, but this is trivial by comparison.

By hiding the instance, `super` violates this fundamental Python idiom for a single role. While that may be comfortable to those accustomed to other OOP languages, it may also qualify as a strike two to others.

Attribute-fetch algorithm

In a *single-inheritance* tree like the preceding example, `super` is straightforward because there's just one obvious follower on the MRO—the superclass of the class containing the `super` call. In fact, in this simple case, the immediate superclass can be had from an instance at `__class__.__bases__[0]` without applying MROs at all:

```
>>> E.__mro__
(<class '__main__.E'>, <class '__main__.C'>, <class 'object'>)
>>> E().__class__.__bases__[0]
<class '__main__.C'>
```

In the more complex class trees of *multiple inheritance*, though, you must understand `super`'s full algorithm to know what it will choose in a given tree.

Here's how this works. The proxy object that `super` returns—created “magically” from runtime info as described in the prior section—uses its saved instance and class containing the call to resolve attribute references as follows:

1. Fetch the MRO of the saved `self` instance's class, available at `self.__class__.__mro__`.
2. Scan this MRO from left to right to find the saved containing class, and skip it.
3. Search the namespace dictionaries of each remaining class in the MRO from left to right until the requested attribute is found.
4. If the attribute was found and is a method, bind it with the saved `self` instance.

This procedure is run for each attribute fetch and is wholly based on MRO ordering. It must start with `self`'s MRO because the containing class's own MRO won't apply if it has been mixed with other classes; class-tree shape and hence MRO may be arbitrary for instances made from lower classes (and may even change dynamically in rare cases).

You can't ignore these underlying mechanics except in very simple class trees. Unlike in Java, the utility of *mix-in* classes in Python makes multiple inheritance from disjoint and independent superclasses a common occurrence in realistic code. And once you add multiple superclasses, you've kicked `super` up to a whole new level.

Universal deployment

Let's illustrate with code. Suppose you've written the following classes that happily deploy `super` in simple single-inheritance mode to implicitly invoke a method one level up from `C`:

```
>>> class A:
    def act(self): print('A')

>>> class B:
    def act(self): print('B')

>>> class C(B):
    def act(self):
        super().act()           # super applied to a single-inheritance tree

>>> C().act()                  # Make an instance and call its method
B
```

If such classes later grow to use more than one superclass, though, `super`'s effects might be surprising—it does not raise an exception when the same name appears in more than one superclass of a multiple inheritance tree but will naively pick just the *leftmost* superclass having the method being run (really, the *first* per the class tree's flattened MRO). This may or may not be the class that you want, and is completely wrong if you want both:

```
>>> class C(B, A):             # Add an A mix-in class with the same method
    def act(self):
        super().act()          # Doesn't fail on conflicts - picks just one

>>> C().act()
B

>>> class C(A, B):
    def act(self):
        super().act()          # If A is listed first, B.act() is no longer run
```



```
>>> C().act()
A
```

This silently masks a source of OOP errors so common that it shows up again in this part’s “Gotchas” ahead. *Explicit* calls are one way to solve this dilemma. With explicit class names, you can choose either the left class, the right class, or both, and with substantially less drama. If you might need to be explicit later, why not use this form earlier too?

```
>>> class C(A, B):           # Explicit form
    def act(self):           # You probably want to be more explicit here
        A.act(self)          # This handles both single and multiple inheritance
        B.act(self)          # So why use the super special case at all?

>>> C().act()
A
B
```

Technically speaking, this example’s explicit *class inheritance* also searches the metaclass type tree for names not defined in superclasses, per “[The Inheritance Bifurcation](#)” on page 823. This secondary-tree search differs from *super*, which always searches just a tail portion of the *instance*’s MRO (and hence just the superclass tree), but is completely moot for defined names and unlikely to matter for undefined names.

The real underlying issue with *super* here, though, is that it requires itself to be called in *every* class’s method in order to propagate the call chain. Without this *universal deployment*, the call dies in the first class that doesn’t call *super*—as in our A and B. While we can add *super* to these classes, too, this is the first of that handful of arduous *super* coding requirements and quickly leads to another issue, as the next section explains.

Call-chain anchors

Since both A and B of the prior section’s example are somewhere on C’s MRO, we might be tempted to make both classes’ methods run by propagating the call with added *super* calls in both.

This scheme is called *cooperative method dispatch*: each class in a tree runs *super* to hand the call off to the next class on the MRO that cares about it. This automatically routes calls through each calling class just once and avoids running a method in a diamond’s common superclass more than once (it appears just once in an MRO). It assumes that the MRO’s order makes sense for your method calls, but explicit class-name calls are a fallback if not.

Armed with that info, here’s the mod in our example:

```
>>> class A:
    def act(self):
        print('A')
        super().act()           # Add a super here too

>>> class B:
    def act(self):
        print('B')
        super().act()           # Add a super here too

>>> class C(B, A):
    def act(self):
        super().act()           # Hope this winds up running all methods

>>> C().act()
B
```

```
A
AttributeError: 'super' object has no attribute 'act'
```

Immediately, though, we're in trouble here, and the reason requires inspecting the MRO of an instance's class:

```
>>> I = C()
>>> I.__class__.__mro__
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

Here's the subtle problem. The `super` in `C` will select `act` in `B`, the next on this MRO; the `super` in `B` will then select `act` in `A`, its follower on `self`'s MRO; but then there are no more `act` definitions to be had: the `super` in `A` looks for `act` in `object` and beyond, and of course fails. We say that there is no *call-chain anchor*—no end point for the call propagation. Hence, the last `super` call in `A` dies with an exception.

In fact, you've just met a possible strike three for this call. While this code works if you omit the `super` in `A`, this policy won't help in general: classes like `A` and `B` are probably designed to be mixed into other classes, too, and it wouldn't make sense to specialize their code just for the `C` class's use case. To propagate `super` method calls, *all* classes must define `super`, too, and there must be an anchor to catch and end the chain somewhere.

Although we can add an anchor in a pointless superclass that defines the method but does not call `super` again, this is another of those arduous coding requirements—and substantially more effort than simply running the explicit class-name calls alternative shown earlier:

```
>>> class X:                                # Code a bogus class, just to appease super
    def act(self):
        print('anchor')

>>> class A: ...same...

>>> class B: ...same...

>>> class C(B, A, X):                        # Add a final anchor, just to appease super
    def act(self):
        super().act()

>>> C().act()
B
A
anchor

>>> [c.__name__ for c in C().__class__.__mro__]
['C', 'B', 'A', 'X', 'object']
```

This works because `X` precedes `object` on the MRO as shown, and hence stops the `act` call chain. Adding the anchor class `X` as a common superclass to both `A` and `B` in a diamond would work, too, because the resulting MRO is the same (remember, the MRO removes all but the last [rightmost] appearance of a class from the DFLR order):

```
>>> class X: ...same...

>>> class A(X): ...same...

>>> class B(X): ...same...

>>> class C(B, A): ...same...

>>> C().act()
```

```
B
A
anchor
```

```
>>> [c.__name__ for c in C().__class__.__mro__]
['C', 'B', 'A', 'X', 'object']
```

Again, though, if you have to code a special class just to appease super, why not just use explicit class names? Similarity to other languages isn't a very good reason, especially in contexts that other languages don't support.

Also, keep in mind that the class selected by super for an attribute reference may not be a superclass at all and may vary per tree that a class is mixed into. For instance, the super in B in our example dispatches to A—the next on the MRO, but a *sibling*, not a superclass. This may be moot in most programs, but if you must be sure that an immediate superclass's method is run, you again must use explicit class names instead of super.

Same argument lists

While super always demands a call-chain anchor, you may occasionally get one for free. As a special case, object defines a *constructor* that can be relied on to anchor some chains (recall that the `__init__` constructor method is a class attribute like any other, despite its odd name and automatic invocation):

```
>>> class A:
    def __init__(self):
        print('A')
        super().__init__()      # Propagate constructor calls

>>> class B:
    def __init__(self):
        print('B')
        super().__init__()      # Propagate constructor calls

>>> class C(B, A):
    def __init__(self):
        super().__init__()      # Assume object anchors constructor chain

>>> I = C()
B
A
```

This propagates the constructor call through C, B, A, and object per the I instance's MRO via cooperative method dispatch as before. This also fails, however, for constructors that take any arguments because that of object takes none except self:

```
>>> class A:
    def __init__(self, name):      # Add an argument to the method
        print('A')
        super().__init__(name)

>>> class B:
    def __init__(self, name):
        print('B')
        super().__init__(name)

>>> class C(B, A):
    def __init__(self, name):
        super().__init__(name)    # But object's arguments list differs
```

```
>>> I = C('Pat')
B
A
TypeError: object.__init__() takes exactly one argument (the instance to initialize)
```

And now you’ve run into another one of those other arduous coding requirements: `super` generally assumes that all the methods in a call chain use the *same arguments* list because the MRO’s ordering of method calls can vary with class-tree shape: an arbitrary change in inheritance may change call order arbitrarily.

This limits flexibility inherently. While you may be able to ensure same arguments for classes used only in a single program and can sometimes fudge it with starred-argument collectors for generality, neither policy will apply to code meant to be reused in multiple contexts—which is really one of the main points behind Python programming.

Noncalls and operator overloading

To close, here are two more `super` oddities. First, keep in mind that `super`, like the MRO, is not just about methods, despite their prevalence in its jargon. It can also fetch the class *data attributes* we met in [Chapter 29](#) and the *bound methods* we met in [Chapter 31](#):

```
>>> class C:
    attr1 = 'hack'           # super also fetches data attributes
    def attr2(self):         # And returns bound methods sans calls
        return 'code'

>>> class D(C):
    def act(self):
        return super().attr1, super().attr2

>>> I = D()
>>> I.act()
('hack', <bound method C.attr2 of <__main__.D object at 0x103993200>>)
>>> I.act()[1]()
'code'
```

When you access a method like `attr2` from a `super` proxy, the proxy binds it with the saved instance to produce a bound method. Fetching a method as a *plain function*, however, may require an explicit class name, like `C.attr2`. This leads down a rabbit hole too deep to plumb here, but it’s another way that `super` clashes with normal semantics.

Second, `super` also doesn’t fully work in the presence of `__X__` operator-overloading methods. If you study the following code, you’ll see that explicit named calls to overloading methods in the superclass work normally, but using the `super` result in an expression fails to dispatch to the superclass’s method:

```
>>> class C:
    def __getitem__(self, ix):    # Indexing overload method
        print('C index')

>>> class D(C):
    def __getitem__(self, ix):    # Redefine to extend here
        print('D index')
        C.__getitem__(self, ix)  # Explicit call form works
        super().__getitem__(ix) # Direct name calls work too
                                # But operators do not! (__getattrattribute__)

>>> I = C()
>>> I[99]
```

```

C index
>>> I = D()
>>> I[99]
D index
C index
C index
TypeError: 'super' object is not subscriptable

```

This behavior is due to the same limitation described in the sidebar “[Delegating Built-ins—or Not](#)” on [page 698](#)—because the proxy object returned by `super` uses the `__getattr__` method we met earlier to catch and dispatch later attribute requests, it fails to intercept the automatic `__X__` method invocations run by built-in operations including expressions, as these begin their search in the class instead of the instance.

This may seem less severe than the other limitations we’ve met, but operators should generally work the same as the equivalent method call, especially for a built-in like this. Not supporting this adds another exception for `super` users to confront and remember. Other languages’ mileage may vary, but in Python, `self` is explicit, multiple-inheritance mix-ins and operator overloading are common, and superclass name changes are rare enough to pass as a red herring.

The super Wrap-Up

So there you have it: a brief tutorial on the `super` built-in, for which you can find copious supplements in all the standard places, some of which seem as focused on defending `super` as on documenting it. Hopefully, the coverage here has given you a balanced view of this tool’s trade-offs while introducing its fundamentals.

As we’ve just seen, in single-inheritance class trees, the `super` call may be used to refer to parent superclasses generically without naming them explicitly. In multiple-inheritance trees, this call can also be used to implement cooperative method dispatch that propagates calls through a tree. The latter role may be especially useful in diamonds, as a conforming method call chain visits each superclass just once.

While these are clear upsides in some contexts, it’s important to know that `super` can also yield highly implicit behavior, which for some programs may not invoke superclasses as expected or required.

To summarize, the `super` method-dispatch technique generally imposes three main coding requirements:

- *Anchors*: the method called by `super` must exist—which requires extra code and calls if no call-chain anchor is present, and mix-in classes can’t be specialized for a single tree’s context.
- *Arguments*: the method called by `super` must have the same argument signature across the entire class tree—which can impair flexibility, especially for implementation-level methods like constructors.
- *Deployment*: every appearance of the method called by `super` but the last must use `super` itself—which can make it difficult to use existing code, change call ordering, override methods, and code self-contained classes.

In addition, `super` builds upon the already complex MRO, can mask problems when single-inheritance trees become multiple-inheritance trees, may select a class other than a superclass in multiple-inheritance trees, and constitutes yet another special case for attribute inheritance, which we’ll revisit in [Chapter 40](#).

In the end, `super` is easy to use and relatively harmless in single-inheritance roles, but its unusual semantics, rigid requirements, and questionable net reward make it a mixed bag. Python programmers, especially those learning Python anew, might be better served by the more general and transparent coding paradigm of explicit class-name references.

But you should judge all this for yourself in an OOP Python program near you.

Class Gotchas

We’ve reached the end of the primary OOP coverage in this book. After exceptions up next, we’ll explore additional class-related examples and topics in the last part of the book, but that part mostly just gives expanded coverage to concepts introduced here. As usual, let’s wrap up this part with the standard warnings about pitfalls to avoid.

Most class issues can be boiled down to namespace issues—which makes sense, given that classes are largely just namespaces with a handful of extra tricks. Some of the items in this section are more like class usage pointers than problems, but even experienced class coders have been known to stumble on a few.

Changing Class Attributes Can Have Side Effects

Theoretically speaking, classes (and class instances) are *mutable* objects. As with built-in lists and dictionaries, you can change them in place by assigning to their attributes—and as with lists and dictionaries, this means that changing a class or instance object may impact multiple references to it.

That’s usually what we want and is how objects change their state in general, but awareness of this issue becomes especially critical when changing *class* attributes. Because all instances generated from a class share the class’s namespace, any changes at the class level are reflected in all instances unless they have their own versions of the changed class attributes.

In Python, we can normally change any attribute in any object to which we have a reference. Consider the following class. Inside the class body, the assignment to the name `a` generates an attribute `X.a`, which lives in the class object at runtime and will be inherited by all of `X`’s instances:

```
>>> class X:
    a = 1          # Class attribute

>>> I = X()
>>> I.a           # Inherited by instance
1
>>> X.a          # Accessible through class
1
```

So far, so good—this is the normal case. But notice what happens when we change the class attribute dynamically *outside* the `class` statement: it also changes the attribute in every object that inherits from the class. Moreover, new instances created from the class during this session or program run also get the dynamically set value, regardless of what the class’s source code says:

```
>>> X.a = 2      # May change more than X
>>> I.a          # I changes too
2
>>> J = X()      # J inherits from X's runtime values
>>> J.a          # (but assigning to J.a changes a in J, not X or I)
2
```

Is this a useful feature or a dangerous trap? You be the judge. As discussed in [Chapter 27](#), you can actually get work done by changing class attributes without ever making a single instance—a technique that can simulate the use of “records” or “structs” in other languages. As a refresher, consider the following unusual but legal Python program:

```
class X: pass                # Make a few attribute namespaces
class Y: pass

X.a = 1                      # Use class attributes as variables
X.b = 2                      # No instances anywhere to be found
X.c = 3
Y.a = X.a + X.b + X.c

for X.i in range(Y.a): print(X.i)  # Prints 0..5
```

Here, the classes `X` and `Y` work like “fileless” modules—namespaces for storing variables we don’t want to clash. This is a perfectly legal Python programming trick, but it’s less appropriate when applied to classes written by others; you can’t always be sure that class attributes you change aren’t critical to the class’s internal behavior. If you’re out to simulate a C struct, you may be better off changing instances than classes, as that way, only one object is affected:

```
class Record: pass
X = Record()
X.name = 'pat'
X.job = 'Pizza maker'
```

Changing Mutable Class Attributes Can Have Side Effects, Too

This gotcha is really an extension of the prior. Because class attributes are shared by all instances, if a class attribute references a *mutable* object, changing that object in place from any instance impacts all instances at once:

```
>>> class C:
    shared = []                # Class attribute
    def __init__(self):
        self.perobj = []      # Instance attribute

>>> x, y = C(), C()          # Two instances
>>> y.shared, y.perobj        # Implicitly share class attrs
([], [])

>>> x.shared.append('hack')   # Impacts y's view too!
>>> x.perobj.append('code')   # Impacts x's data only
>>> x.shared, x.perobj
(['hack'], ['code'])

>>> y.shared, y.perobj        # y sees change made through x
(['hack'], [])
>>> C.shared                  # Stored on class and shared
['hack']
```

This effect is no different than many we’ve seen in this book already: mutable objects are shared by simple variables, globals are shared by functions, module-level objects are shared by multiple importers, and mutable function arguments are shared by the caller and the callee. All of these are cases of general behavior—multiple references to a mutable object—and all are impacted if the shared object is changed in place from any reference.

Here, this occurs in class attributes shared by all instances via inheritance, but it's the same phenomenon at work. It may be made more subtle by the different behavior of assignments to instance attributes themselves:

```
x.shared.append('hack')    # Changes shared object attached to class - in place
x.shared = 'hack'         # Changed or creates instance attribute attached to x
```

But again, this is not a problem, it's just something to be aware of; shared mutable class attributes can have many valid uses in Python programs.

Multiple Inheritance: Order Matters

This may be obvious by now, but it's worth underscoring one last time: if you use multiple inheritance, the order in which superclasses are listed in the `class` statement header can be critical. Python always searches superclasses from left to right, according to their order in the header line.

For instance, in the multiple inheritance example we studied in [Chapter 31](#), imagine that the `Super` class implemented a `__str__` method, too:

```
class ListTree:
    def __str__(self): ...

class Super:
    def __str__(self): ...

class Sub(ListTree, Super):    # Get ListTree's __str__ by listing it first

x = Sub()                    # Inheritance searches ListTree before Super
```

Which class would we inherit it from—`ListTree` or `Super`? As inheritance searches generally proceed from left to right, we would get the method from whichever class is listed first (leftmost) in `Sub`'s `class` header. Presumably, we would list `ListTree` first because its whole purpose is its custom `__str__`. Indeed, we had to do this in [Chapter 31](#) when mixing this class with a `tkinter.Button` that had a `__str__` of its own.

But now, suppose `Super` and `ListTree` have their own versions of other same-named attributes, too. If we want one name from `Super` and another from `ListTree`, the order in which we list them in the class header won't help—we will have to override inheritance by manually assigning to the attribute name in the `Sub` class:

```
class ListTree:
    def __str__(self): ...
    def other(self): ...

class Super:
    def __str__(self): ...
    def other(self): ...

class Sub(ListTree, Super):    # Get ListTree's __str__ by listing it first
    other = Super.other       # But explicitly pick Super's version of other
    def __init__(self):
        ...

x = Sub()                    # Inheritance searches Sub before ListTree/Super
```


Here, the assignment to `other` within the `Sub` class creates `Sub.other`—a reference back to the `Super.other` object. Because it is lower in the tree, `Sub.other` effectively hides `ListTree.other`, the attribute that the inheritance search would normally find. Similarly, if we listed `Super` first in the class header to pick up its `other`, we would need to select `ListTree`’s method explicitly:

```
class Sub(Super, ListTree):           # Get Super's other by order
    __str__ = ListTree.__str__       # Explicitly pick ListTree.__str__
```

For another example of the technique shown here in action, see the discussion of explicit conflict resolution in [“Attribute Conflict Resolution” on page 801](#). Ultimately, multiple inheritance is an advanced tool. Even if you understood the last paragraph, it’s still a good idea to use it sparingly and carefully. Otherwise, the meaning of a name may come to depend on the order in which classes are mixed in an arbitrarily far-removed subclass.

As a rule of thumb, multiple inheritance works best when your mix-in classes are as self-contained as possible—because they may be used in a variety of contexts, they should not make assumptions about names related to other classes in a tree. The pseudoprivate `__X` attributes feature we studied in [Chapter 31](#) can help by localizing names that a class relies on owning and limiting the names that mix-in classes add to the mix. In this example, for instance, if `ListTree` only means to export its custom `__str__`, it can name its `other` method `__other` to avoid clashing with like-named classes in the tree.

Scopes in Methods and Classes

When working out the meaning of names in class-based code, it helps to remember that classes introduce local scopes, just as functions do, and methods are simply further nested functions. In the following example, the `generate` function returns an instance of the nested `Hack` class. Within its code, the class name `Hack` is assigned in the `generate` function’s local scope and hence is visible to any further nested functions, including code inside `method`; it’s in the *E* enclosing-function layer of the LEGB scope lookup rule:

```
def generate():
    class Hack:                # Hack is a name in generate's local scope
        count = 1
        def method(self):
            print(Hack.count)   # Visible in generate's scope, per LEGB rule (E)
    return Hack()

generate().method()
```

This example works because the local scopes of all enclosing function defs are automatically visible to nested defs—including nested method defs, as in this example.

Even so, keep in mind that method defs cannot see the local scope of the enclosing *class*; they can see only the local scopes of enclosing defs. That’s why methods must go through the `self` instance or the class name to reference methods and other attributes defined in the enclosing `class` statement. For example, code in the method must use `self.count` or `Hack.count`, not just `count`.

To avoid nesting, we could restructure this code such that the class `Hack` is defined at the top level of the module: the nested `method` function and the top-level `generate` will then both find `Hack` in their global scopes; it’s not localized to a function’s scope, but is still local to a single module:

```
def generate():
    return Hack()

class Hack:
    count = 1
    def method(self):
        print(Hack.count)

generate().method()
```

Code tends to be simpler in general if you avoid nesting classes and functions. On the other hand, class nesting is useful in *closure* contexts, where the enclosing function's scope retains *state* used by the class or its methods. In the following, the nested `method` has access to its own scope, the enclosing function's scope (for `label`), the enclosing module's global scope, anything saved in the `self` instance by the class, and the class itself via its nonlocal name:

```
>>> def generate(label):
    class Hack:
        count = 1
        def method(self):
            print(f'{label}={Hack.count}')
    return Hack

>>> aclass = generate('Gotchas')
>>> I = aclass()
>>> I.method()
Gotchas=1
```

Miscellaneous Class Gotchas

Here's a handful of additional class-related warnings, mostly as review:

- **Choose per-instance or class storage wisely.** On a similar note, be careful when you decide whether an attribute should be stored on a class or its instances: the former is shared by all instances, and the latter will differ per instance. In a GUI program, for instance, if you want information to be shared by all of the window class objects your application will create (e.g., the last directory used for a Save operation or an already entered password), it might be best stored as class-level data; if stored in the instance as `self` attributes, it will vary per window or be missing entirely when looked up by inheritance.
- **You usually want to call superclass constructors.** Remember that Python runs only one `__init__` constructor method when an instance is made—the first it finds by inheritance. It does not automatically run the constructors of all superclasses higher up. Because constructors normally perform required startup work, you'll usually need to run a superclass constructor from a subclass constructor—using either an explicit call through the superclass's name or `super`, passing along whatever arguments are required—unless you mean to replace the super's constructor altogether, or the superclass doesn't have or inherit a constructor at all.
- **Stay tuned for a fix for `__getattr__` and built-ins.** Another reminder: as noted in [Chapter 28](#) and elsewhere, classes that use the `__getattr__` operator-overloading method to delegate attribute fetches to wrapped objects may fail unless operator-overloading methods are redefined in the wrapper class. The names of operator-overloading methods implicitly fetched by built-in operations are not routed through generic attribute-interception methods. To work around this, you must redefine such methods in wrapper classes, either manually, with tools, or by definition in superclasses; you'll learn how in [Chapter 39](#).

“Overwrapping-itis”

Finally, when used well, the code reuse features of OOP make it excel at cutting development time. Sometimes, though, OOP’s abstraction potential can be abused to the point of making code difficult to understand. If classes are layered too deeply, code can become obscure; you may have to search through many classes to discover what an operation does.

Imagine, for example, a framework with hundreds of classes and a dozen levels of inheritance (this is a true story, but details have been omitted to protect the innocent). Deciphering method calls in such a complex system may be a monumental task: multiple classes might have to be consulted for even the most basic of operations. In fact, the logic of such a system can be so deeply wrapped that understanding a piece of code in some cases may require days of wading through related files. This obviously isn’t ideal for programmer productivity.

The most general rule of thumb of Python programming applies here, too: *don’t make things complicated unless they truly must be*. Wrapping your code in multiple layers of classes to the point of incomprehensibility is always a bad idea. Abstraction is the basis of polymorphism and encapsulation, and it can be a very effective tool when used well. However, you’ll simplify debugging and aid maintainability if you make your class interfaces intuitive, avoid making your code overly abstract, and keep your class hierarchies short and small unless there is a good reason to do otherwise. Remember: code you write is generally code that others must read.

Chapter Summary

This chapter presented an assortment of class-related topics, including subclassing built-in types, the relationship of types and classes, slots, properties, static methods, decorators, and `super`. Most are optional extensions to the OOP toolbox in Python but may become more useful as you start writing larger object-oriented programs, and all are fair game if they appear in code you must understand. As noted earlier, some of these topics are continued in the final part of this book; be sure to look ahead for more info on properties, descriptors, decorators, and metaclasses.

This is the end of the class part of this book, so you’ll find the usual lab exercises at the end of the chapter: be sure to work through them to get some practice coding real classes. In the next chapter, we’ll begin our look at our last core language topic, *exceptions*—Python’s mechanism for communicating errors and other conditions to your code. This is a relatively lightweight topic but it was saved for last because new exceptions must be coded as classes. Before we tackle that final core subject, though, take a look at this chapter’s quiz and the lab exercises.

Test Your Knowledge: Quiz

1. Name two ways to extend a built-in object type.
2. What are function and class decorators used for?
3. How are normal and static methods different?
4. Are tools like `__slots__` and `super` valid to use in your code?

Test Your Knowledge: Answers

1. You can embed a built-in object in a wrapper class, or subclass the built-in type directly. The latter approach tends to be simpler, as most original behavior is automatically inherited. This works because types are classes, though the way you create an instance from a type/class determines its functionality.
2. Function decorators are generally used to manage a function or method or add to it a layer of logic that is run each time the function or method is called. They can be used to log or count calls to a function, check its argument types, and so on. They are also used to “declare” static methods (simple functions in a class that are not passed an instance, however they are called), as well as class methods and properties. Class decorators are similar but manage whole objects and their interfaces instead of a function call.
3. Normal (instance) methods receive a `self` argument (the implied instance), but static methods do not. Static methods are simple functions nested in class objects. To make a method static, it must either be run through a special built-in function or be decorated with decorator syntax. Python also allows simple functions in a class to be called through the class without this step, but calls through instances still require static-method declaration.
4. *Of course*, but you shouldn’t use advanced tools automatically without carefully considering their implications. Slots, for example, can break code; `super` can mask later problems when used for single inheritance, and in multiple inheritance brings with it substantial complexity for an isolated use case; and both require universal deployment to be most useful. Evaluating new or advanced tools is a primary task of any engineer, and this is why we explored trade-offs in this chapter. This book’s goal is not to tell you which tools to use but to underscore the importance of objectively analyzing them—a task often given too low a priority in the software field. In engineering, as in life in general, we shouldn’t let other people make choices for us.

Test Your Knowledge: Part VI Exercises

These exercises ask you to write a few classes and experiment with some existing code. Of course, the problem with existing code is that it must be existing. To work with the `set` class in exercise 5, either copy/paste [Example 32-1](#) from [emedia](#), find it in this book’s examples package (see the [Preface](#) for pointers), or type it up by hand (mildly tedious but a great way to make syntax more concrete). These programs are growing sophisticated, so be sure to check the solutions at the end of the book for pointers. You’ll find them in [Appendix B](#), under “[Part VI, Classes and OOP](#)” on page 1181.

1. *Inheritance*: Write a class called `Adder` that exports a method `add(self, x, y)`, which prints a “Not Implemented” message. Then, define two subclasses of `Adder` that implement the `add` method:

`ListAdder`

With an `add` method that returns the concatenation of its two list arguments

`DictAdder`

With an `add` method that returns a new dictionary containing the items in both its two dictionary arguments (any definition of dictionary addition will do; see dictionary union in [Chapter 8](#) for tips)

Experiment by making instances of all three of your classes interactively and calling their `add` methods.

Now, extend your `Adder` superclass to save an object in the instance with a constructor (e.g., assign `self.data` a list or a dictionary), and overload the `+` operator with an `__add__` method to automatically dispatch to your `add` methods (e.g., `X + Y` triggers `X.add(X.data,Y)`). Where is the best place to put the constructors and operator-overloading methods (i.e., in which classes)? What sorts of objects can you add to your class instances?

In practice, you might find it easier to code your `add` methods to accept just one real argument (e.g., `add(self,y)`) and add that one argument to the instance's current data (e.g., `self.data + y`). Does this make more sense than passing two arguments to `add`? Would you say this makes your classes more “object-oriented”?

2. *Operator overloading*: Write a class called `MyList` that shadows (“wraps”) a Python list: it should overload most list operators and operations, including `+`, indexing, iteration, slicing, and list methods such as `append` and `sort`. See the Python reference manual or other documentation for a list of all possible methods to support. Also, provide a constructor for your class that takes an existing list (or a `MyList` instance) and copies its components into an instance attribute. Experiment with your class interactively. Things to explore:
 - Why is copying the initial value important here?
 - Can you use an empty slice (e.g., `start[:]`) to copy the initial value if it's a `MyList` instance?
 - Is there a general way to route list method calls to the wrapped list?
 - Can you add a `MyList` and a regular list? How about a list and a `MyList` instance?
 - What type of object should operations like `+` and slicing return? What about indexing operations?
 - You may implement this sort of wrapper class by embedding a real list in a standalone class or by extending the built-in list type with a subclass. Which is easier, and why?
3. *Subclassing*: Make a subclass of `MyList` from exercise 2 called `MyListSub`, which extends `MyList` to print a message to `stdout` before each call to the `+` overloaded operation and counts the number of such calls. `MyListSub` should inherit basic method behavior from `MyList`. Adding a sequence to a `MyListSub` should print a message, increment the counter for `+` calls, and perform the superclass's method. Also, introduce a new method that prints the operation counters to `stdout` (i.e., your console window) and experiment with your class interactively. Do your counters count calls per instance or per class (for all instances of the class)? How would you program the other option? (Hint: it depends on which object the count members are assigned to: class members are shared by instances, but `self` members are per-instance data.)
4. *Attribute methods*: Write a class called `Attrs` with methods that intercept every attribute qualification (both fetches and assignments), and print messages listing their arguments to `stdout`. Create an `Attrs` instance and experiment with qualifying it interactively. What happens when you try to use the instance in expressions? Try adding, indexing, and slicing the instance of your class. (Note: a fully generic approach based upon `__getattr__` requires [Chapter 39](#)'s workarounds for reasons noted in [Chapter 28](#) and later and summarized in the solution to this exercise.)

5. *Set objects*: Experiment with the set class of [Example 32-1](#) and described in “[Extending Types by Embedding](#)” on page 818. Run commands to do the following sorts of operations:
 - Create two sets of integers, and compute their intersection and union by using & and | operator expressions.
 - Create a set from a string, and experiment with indexing your set. Which methods in the class are called?
 - Try iterating through the items in your string set using a for loop. Which methods run this time?
 - Try computing the intersection and union of your string set and a simple Python string. Does it work?
 - Now, extend your set by subclassing to handle arbitrarily many operands using the *args argument form. (Hint: see the function versions of these algorithms in [Chapter 18](#).) Compute intersections and unions of multiple operands with your set subclass. How can you intersect three or more sets, given that & has only two sides?
 - How would you go about emulating other list operations in the set class? (Hint: __add__ can catch concatenation, and __getattr__ can pass most named list method calls like append to the wrapped list.)
6. *Class tree links*: In “[Namespaces: The Conclusion](#)” on page 724 and in “[Multiple Inheritance and the MRO](#)” on page 796, we learned that classes have a __bases__ attribute that returns a tuple of their superclass objects (the ones listed in parentheses in the class header). Use __bases__ to extend any or all three of the listing mix-in classes we wrote in [Chapter 31](#) so that they print the names of the immediate superclasses of the instance’s class. Modding [Example 31-10](#) first may be easiest. When you’re done, the first line of the string representation should look like this (your hex addresses will almost certainly vary):

```
<Instance of Sub(Super, Lister), address 0x...:
```

7. *Composition*: Simulate a fast-food ordering scenario by defining four classes:

Lunch

A container and controller class

Customer

The actor who buys food

Employee

The actor from whom a customer orders

Food

What the customer buys

To get you started, here are the classes and methods you’ll be defining:

```
class Lunch:
    def __init__(self)                # Make/embed Customer and Employee
    def order(self, foodName)         # Start a Customer order simulation
    def result(self)                  # Ask the Customer what Food it has

class Customer:
    def __init__(self)                # Initialize my food to None
    def placeOrder(self, foodName, employee) # Place order with an Employee
    def printFood(self)               # Print the name of my food
```

```
class Employee:
    def takeOrder(self, foodName)    # Return a Food, with requested name

class Food:
    def __init__(self, name)        # Store food name
```

The order simulation should work as follows:

- The Lunch class’s constructor should make and embed an instance of Customer and an instance of Employee, and it should export a method called order. When called, this order method should ask the Customer to place an order by calling its placeOrder method. The Customer’s placeOrder method should, in turn, ask the Employee object for a new Food object by calling Employee’s takeOrder method.
- Food objects should store a food name string (e.g., “burritos”), passed down from Lunch.order, to Customer.placeOrder, to Employee.takeOrder, and finally to Food’s constructor. The top-level Lunch class should also export a method called result, which asks the customer to print the name of the food it received from the Employee via the order (this can be used to test your simulation).

Note that Lunch needs to pass either the Employee or itself to the Customer to allow the Customer to call Employee methods.

Experiment with your classes interactively by importing the Lunch class, calling its order method to run an interaction, and then calling its result method to verify that the Customer got what it ordered. If you prefer, you can also simply code test cases as self-test code in the file where your classes are defined, using the module `__name__` trick of [Chapter 25](#). In this simulation, the Customer is the active agent; how would your classes change if Employee were the object that initiated customer/employee interaction instead?

8. *Zoo animal hierarchy*: Consider the class tree shown in [Figure 32-1](#).

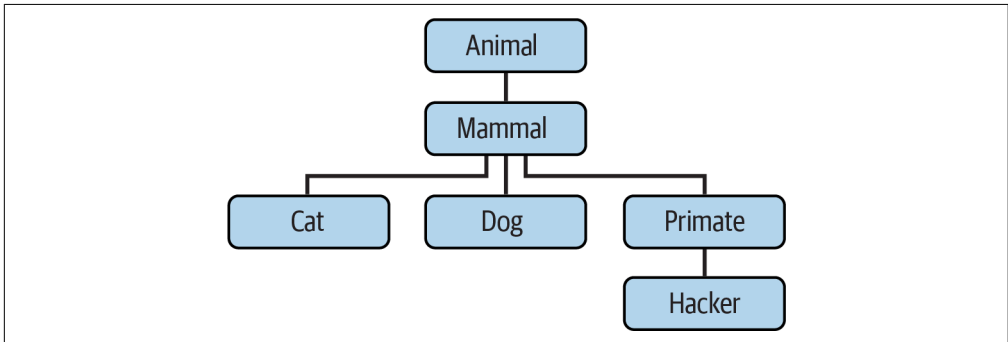


Figure 32-1. A zoo hierarchy composed of classes linked into an inheritance tree

Code a set of six class statements to model this taxonomy with Python *inheritance*. Then, add a `speak` method to each of your classes that prints a unique message and a `reply` method in your top-level `Animal` superclass that simply calls `self.speak` to invoke the category-specific message printer in a subclass below (this will kick off an independent inheritance search from `self`). Finally, remove the `speak` method from your `Hacker` class so that it picks up the default above it. When you’re finished, your classes should work this way:

```

$ python3
>>> from zoo import Cat, Hacker
>>> spot = Cat()
>>> spot.reply()                # Animal.reply: calls Cat.speak
meow
>>> data = Hacker()            # Animal.reply: calls Primate.speak
>>> data.reply()
Hello world!

```

Why You Will Care: OOP by the Masters

Almost invariably, when teaching live Python classes, about halfway through the OOP section, people who have used OOP in the past are following along intensely, while people who have not are beginning to glaze over (or nod off completely). The point behind the technology just isn't apparent.

A book like this has the luxury of slowly presenting material like the overview in [Chapter 26](#), and the gradual tutorial of [Chapter 28](#)—in fact, you should probably review those sections again if you're starting to feel like OOP is just some computer science mumbo-jumbo. Though OOP adds more structure than the generators we met earlier, it similarly relies on some magic (inheritance search and a special first argument) that beginners can understandably find difficult to rationalize.

In real classes, however, to help get the newcomers on board (and keep them awake), it often helps to stop and ask the experts in the audience why they use OOP. The answers they've given might help shed some light on the purpose of OOP if you're new to the subject.

Here, then, with only a few embellishments, are the most common reasons to use OOP, as cited by students over the years:

Code reuse

This one's easy and is the main reason for using OOP. By supporting inheritance, classes make it natural to program by customization instead of starting each project from scratch.

Encapsulation

Wrapping up implementation details behind object interfaces insulates users of a class from code changes.

Structure

Classes provide new local scopes, which minimizes name clashes. They also provide a natural place to write and look for implementation code and to manage object state.

Maintenance

Classes naturally promote code factoring, which allows us to minimize redundancy. Thanks to both the structure and code reuse support of classes, usually only one copy of the code needs to be changed.

Consistency

Classes and inheritance allow you to implement common interfaces and hence create a common look and feel in your code; this eases debugging, comprehension, and maintenance.

Polymorphism

This is more a property of OOP than a reason for using it, but by supporting code generality, polymorphism makes code more flexible and widely applicable and hence more reusable.

Other

And, of course, the number one reason students gave for using OOP: it looks good on a résumé! (OK, this one was added as a joke, but it is important to be familiar with OOP if you plan to work in the software field today.)

Finally, keep in mind the guidance given multiple times in this part of this book: you won't fully appreciate OOP until you've used it for a while. Pick a project, study larger examples, work through the exercises. Do whatever it takes to get your feet wet with OOP code. It's optional stuff and may even be overkill in some contexts, but it's generally worth the effort.

Exceptions

Exception Basics

This part of the book deals with *exceptions*—events that signal conditions and modify the flow of control through a program. In Python, exceptions are triggered automatically on errors, and they can be both triggered and intercepted by your code. They are processed by four statements we’ll study here, the first of which comes in multiple flavors that qualify as different statement forms by some measures:

`try/except/else/finally`

Catch and recover from exceptions raised by Python, or by you

`raise`

Trigger an exception manually in your code

`assert`

Conditionally trigger an exception in your code

`with`

Use context managers that automate exception handling

We’ve met some of these briefly before, but full coverage of this topic was saved until the end of the main part of this book because you need to know about *classes* to code exceptions of your own. Still, with a few exceptions (pun intended), you’ll find that exception handling is simple in Python because it’s integrated into the language itself as another high-level tool. Before we dig into the “how,” though, let’s get clear on the “why.”

Why Use Exceptions?

In a nutshell, exceptions let us jump out of arbitrarily large chunks of a program. Consider the hypothetical pizza-making robot we discussed earlier in the book. Suppose we took the idea seriously and actually built such a machine. To make a pizza, our culinary automaton would need to execute a plan, which we would implement as a Python program: it would take an order, prepare the dough, add toppings, bake the pie, and so on.

Now, suppose that something goes very wrong during the “bake the pie” step. Perhaps the oven is broken, or perhaps our robot miscalculates its reach and spontaneously combusts. Clearly, we want to be able to jump to code that handles such unusual states quickly. As we have no hope of finishing the pizza task in such unusual cases, we might as well abandon the entire plan.

That's exactly what exceptions let your programs do: they can jump to an exception handler in a single step, abandoning all activity begun since the exception handler was entered. Code in the exception handler can then respond to the raised exception as appropriate (by calling the fire department, for instance!).

One way to think of an exception is as a sort of structured “go-to.” An *exception handler* (try statement) leaves a marker and executes some code. Somewhere further ahead in the program, an exception is raised that makes Python jump back to that marker, abandoning any code that was started and functions that were called after the marker was left. The net effect unwinds the program's control flow back to the marker and resumes there.

This protocol provides a coherent way to respond to unusual events. Moreover, because Python jumps to the handler statement immediately, your code is simpler—there is usually no need to check status codes after every operation and function call that could possibly fail. Instead, we catch errors only where we need to recover from them.

Exception Roles

In less hypothetical programs, exceptions serve a variety of purposes. Here are some of their most common roles:

Error handling

Python raises exceptions whenever it detects errors in programs at runtime. You can catch and respond to the errors in your code, or ignore the exceptions that are raised. If an error is ignored, Python's default exception-handling behavior kicks in: it stops the program and prints an error message. If you don't want this default behavior, code a try statement to catch and recover from the exception—Python will jump to your try handler when the error is detected in the statement's code, and your program will resume execution after the try.

Event notification

Exceptions can also be used to signal valid conditions without you having to pass result flags around a program or test them explicitly. For instance, a search routine might raise an exception on failure, rather than returning an integer result code—and hoping that the code will never be a valid result.

Special-case handling

Sometimes a condition may occur so rarely that it's hard to justify convoluting your code to handle it in multiple places. You can often eliminate special-case code by handling unusual cases in exception handlers in higher levels of your program. An `assert` can similarly be used to check that conditions are as expected during development.

Termination actions

As you'll see, the `finally` option in a try statement allows you to guarantee that required closing-time operations will be performed, regardless of the presence or absence of exceptions in your programs. The `with` statement offers an alternative in this department for objects that support its expected method-call protocol.

Unusual control flows

Finally, because exceptions are a sort of high-level and structured “go-to,” you can use them as the basis for implementing exotic control flows. For instance, although the language does not explicitly support backtracking, you can implement it in Python by using exceptions and logic to unwind

assignments.¹ There is no “go to” statement in Python (thankfully) and no built-in backtracking (today), but exceptions can sometimes serve similar roles; a `raise`, for instance, can be used to jump out of multiple loops in ways that `break` cannot.

We saw some of these roles briefly earlier and will study typical exception use cases in action later in this part of the book. For now, let’s get started with a look at Python’s exception-processing tools.

Exceptions: The Short Story

Compared to some other core language topics we’ve explored in this book, exceptions are a fairly light-weight tool in Python. Because they are so simple, let’s jump right into some code.

Default Exception Handler

Suppose we write the following function in the interactive REPL of our choice:

```
>>> def fetcher(obj, index):
    return obj[index]
```

There’s not much to this function—it simply indexes an object on a passed-in index. In normal operation, it returns the result of a legal index:

```
>>> food = 'pizza'
>>> fetcher(food, 4)           # Like x[4], last item
'a'
```

However, if we ask this function to index off the end of the string, an exception will be triggered when the function tries to run `obj[index]`. Python detects out-of-bounds indexing for sequences and reports it by *raising* (triggering) the built-in `IndexError` exception:

```
>>> fetcher(food, 5)           # Default handler - console interface
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    fetcher(food, 5)
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

Because our code does not explicitly catch this exception, it filters back up to the top level of the program and invokes the *default exception handler*, which simply prints the standard error message shown here.

By this point in the book, you’ve probably seen your share of standard error messages. They include the exception that was raised, along with a *stack trace*—a list of all the lines and functions that were active when the exception occurred, which has been largely omitted in this book for space and brevity.

The error message text here was printed by Python 3.12 in a console. It can vary slightly per release, and even per interactive REPL, so you shouldn’t rely upon its exact form—in either this book or your code. When you’re coding interactively in a console interface, the filename may be just “<stdin>,” meaning the standard input stream.

¹ For any computer scientists in the audience, true backtracking is not part of the Python language. Backtracking undoes computations before it jumps back, but Python exceptions do not: local variables in open function calls run by the `try` are simply discarded, but changes made to globals and objects are retained (see the demo ahead). Even the generator functions and expressions we met in [Chapter 20](#) don’t do full backtracking—they simply respond to `next(G)` requests by restoring saved state and resuming. For more on backtracking, try books on AI or the Prolog or Icon programming languages.

When working in the IDLE GUI's interactive shell today, though, the filename is "<pyshell...>," and source lines are displayed, too. Either way, file line numbers are not very meaningful when there is no file (you'll see more interesting error messages later in this part of the book):

```
>>> fetcher(food, 5)                                # Default handler - IDLE GUI interface
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    fetcher(food, 5)
  File "<pyshell#0>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range
```

In a more realistic program launched outside the interactive REPL, after printing an error message the default handler at the top also *terminates* the program immediately. That course of action makes sense for simple scripts; errors often should be fatal, and the best you can do when they occur is inspect the standard error message.

Catching Exceptions

Sometimes, though, program termination on exceptions isn't what you want. Server programs, for instance, typically need to remain active even after internal errors. If you don't want the default exception behavior, wrap the call in a try statement to catch exceptions yourself (copy/pasters: omit the "... here per the note ahead):

```
>>> try:
...     fetcher(food, 5)
... except IndexError:                                # Catch and recover
...     print('got exception')
...
got exception
>>>
```

Now, Python automatically jumps to your *handler*—the block under the except clause that names the exception raised—when an exception is triggered while the try block is running. The net effect is to wrap a nested block of code in an error handler that intercepts the block's exceptions.

When working interactively like this, after the except clause runs, we wind up back at the Python prompt. In a more realistic program, try statements not only catch exceptions but also *recover* from them:

```
>>> def catcher():
...     try:
...         fetcher(food, 5)
...     except IndexError:
...         print('got exception')                    # Catch and recover more
...         print('continuing')

>>> catcher()
got exception
continuing
>>>
```

This time, after the exception is caught and handled, the program resumes execution after the entire try statement that caught it—which is why we get the "continuing" message here. We don't see the standard error message, and the program continues on its way normally.

Notice, though, that there's no way in Python to *go back* to the code that triggered the exception (short of rerunning the code that reached that point all over again, of course). Once you've caught the exception, control continues after the entire `try` that caught the exception, not after the statement that kicked off the exception. In fact, Python clears the memory of any functions that were exited as a result of the exception, like `fetcher` in our example; their variables are discarded, and they're not resumable. The `try` both catches exceptions and is where the program resumes.

Python does not, however, *undo* any work done by the `try` block up to the point where the exception occurred—any changes made to referenced mutable objects and accessible global names live on. This isn't a problem if it's known:

```
>>> L, S = [], 'text'
>>> def modder():
    L.append('added')           # Change a mutable
    global S; S = 'changed'    # Change a global
    fetcher(food, 5)           # Trigger an exception

>>> try:
...     modder()
... except IndexError:
...     print('got exception')
...
got exception
>>> L, S
(['added'], 'changed')      # Changes retained
```

As you'll see later in this part of the book, the `try` can also use `except*` clauses to process *multiple* exceptions, but this is a convoluted extension with narrow scope that doesn't play well with others, and you can safely defer studying until you've mastered the fundamentals.



Presentation note: The interactive REPL's “...” continuation prompt reappears in this part for some top-level `try` statements, because their code won't work if copied and pasted unless nested in a function or class (the `except` and other lines must align with the `try`, and not have extra preceding spaces that are needed to illustrate their indentation structure here). To run, simply type or paste statements with “...” prompts one line at a time, and without their leading “...” prompts.

Raising Exceptions

So far, we've been letting Python raise exceptions for us by making mistakes (on purpose this time!), but our scripts can raise exceptions too—that is, exceptions can be raised by Python or by your program, and can be caught or not. To trigger an exception manually, simply run a `raise` statement. User-triggered exceptions are caught the same way as those Python raises. The following may not be the most useful Python code ever penned, but it makes the point—raising the built-in `IndexError` exception:

```
>>> try:
...     raise IndexError
... except IndexError:
...     print('got exception')
...
got exception
```

As usual, if they're not caught, user-triggered exceptions are propagated up to the top-level default exception handler and terminate the program with a standard error message:

```
>>> raise IndexError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError
```

As you'll see in the next chapter, the `assert` statement can be used to trigger exceptions, too—it's a conditional `raise` predicated on a test, used mostly for debugging purposes and sanity checks during development:

```
>>> assert 1 < 0, 'Not in this universe!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Not in this universe!
```

Also in the next chapter, you'll learn that `raise` can use a `from` clause to “chain” exceptions; generally speaking, this is not common, but can be used to give more context where it's useful.

User-Defined Exceptions

The `raise` statement demos in the prior section raised `IndexError`, a *built-in* exception defined in Python's built-in scope. As you'll learn later in this part of the book, you can also define new exceptions of your own that are specific to your programs. User-defined exceptions are coded with *classes*, which inherit from a built-in exception class—usually, the class named `Exception`:

```
>>> class Combust(Exception): pass          # User-defined exception

>>> def makePizza():
    raise Combust()                        # Raise an instance

>>> try:
...     makePizza()
... except Combust:                        # Catch class name
...     print('got exception')
...
got exception
>>>
```

As you'll see in upcoming chapters, exception classes allow scripts to build exception categories, which can inherit behavior and have attached state information and methods, and an `as` clause on an `except` can gain access to the exception object itself. Exception classes can also customize their message text displayed if they're not caught:

```
>>> class Combust(Exception):
    def __str__(self):
        return 'Call the fire department!...'

>>> raise Combust()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Combust: Call the fire department!...
>>>
```

Termination Actions

Finally, try statements can say “finally”—that is, they may include `finally` blocks. These look like except handlers for exceptions, but the `try/finally` combination specifies termination actions that always execute “on the way out,” regardless of whether an exception occurs in the `try` block or not. Continuing our REPL session:

```
>>> try:
...     fetcher(food, 4)
...     finally:
...         print('after fetch')
...
... 'a'
after fetch
>>>
```

Here, if the `try` block finishes *without* an exception, the `finally` block will run, and the program will resume after the entire `try`. In this case, this statement seems a bit silly—we might as well have simply typed the `print` right after a call to the function, and skipped the `try` altogether:

```
fetcher(food, 4)
print('after fetch')
```

There is a problem with coding this way, though: if the function call raises an exception, the `print` will never be reached. The `try/finally` combination avoids this pitfall—when an exception *does* occur in a `try` block, `finally` blocks are executed while the program is being unwound:

```
>>> def after():
...     try:
...         fetcher(food, 5)
...     finally:
...         print('after fetch')
...         print('after try?')
...                                     # Always run
...                                     # Run only if no exception

>>> after()
after fetch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in after
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
>>>
```

Here, we don’t get the “after try?” message because control does not resume after the `try` statement when an exception occurs. Instead, Python jumps back to run the `finally` action and then *propagates* the exception up to a prior handler (in this case, to the default handler at the top). If we change the call inside this function so as not to trigger an exception, the `finally` code still runs, but the program continues after the `try`:

```
>>> def after():
...     try:
...         fetcher(food, 4)
...     finally:
...         print('after fetch')
...         print('after try?')
...                                     # Both run if no exception

>>> after()
after fetch
```

```
after try?  
>>>
```

In practice, `except` clauses in a `try` are useful for catching and recovering from exceptions, and `finally` clauses come in handy to guarantee that termination actions will fire regardless of any exceptions that may occur in the `try` block's code. For instance, you might use `try/except` combinations to catch errors raised by code that you import from a third-party library, and `try/finally` combos to ensure that calls to close files or terminate server connections are always run. We'll code some such practical examples later in this part of the book.

Although they serve conceptually distinct purposes, you can also mix `except` and `finally` clauses in the *same* `try` statement—the `finally` is run on the way out regardless of whether an exception was raised, and regardless of whether the exception was caught by an `except` clause. Such combos have rules you'll meet in the next chapter.

As you'll also learn in the next chapter, Python provides an alternative to the `try/finally` mix when using some types of built-in and user-defined objects. The `with` statement runs a *context manager* object's methods to guarantee that termination actions occur, irrespective of any exceptions in its nested block:

```
>>> with open('pizzarobot.txt', 'w') as file:           # Always close file on exit  
    file.write('Catch fire!\n')
```

Although this option requires fewer lines of code, it's applicable only when processing certain object types, so `try/finally` is a more general termination structure, and is often simpler than coding a class in cases where `with` is not already supported. On the other hand, `with` may also run startup actions too, and supports user-defined context management code with access to Python's full OOP toolset. To see how, let's move on to the next chapter.

Chapter Summary

And that is the majority of the exception story; exceptions really are a simple tool.

To summarize, Python exceptions are a control-flow device. They may be raised by Python, or by your own programs. In both cases, they may be ignored (to trigger the default error handler), or caught by `try` statements (to be processed by your code). The `try` statement comes in logically distinct forms that can be combined—one that handles exceptions, and one that runs finalization code regardless of whether exceptions occur or not. Python's `raise` and `assert` statements trigger exceptions on demand—both built-ins and new exceptions we define with classes—and the `with` statement is an alternative way to ensure that termination actions are carried out for objects that support it.

In the rest of this part of the book, we'll fill in some of the details about the statements involved, examine the other sorts of clauses that can appear under a `try` (spoiler: it also allows an `else` for the no-exception case), and discuss class-based exception objects. The next chapter begins our tour by taking a closer look at the statements we introduced here. Before you turn the page, though, here are a few quiz questions to review.

Test Your Knowledge: Quiz

1. Name three things that exception processing is good for.
2. What happens to an exception if you don't do anything special to handle it?
3. How can your script recover from an exception?
4. Name two ways to trigger exceptions in your script.
5. Name two ways to specify actions to be run at termination time, whether an exception occurs or not.

Test Your Knowledge: Answers

1. Exception processing is useful for error handling, termination actions, and event notification. It can also simplify the handling of special cases and can be used to implement alternative control flows as a kind of structured “go-to” operation. In general, exception processing also cuts down on the amount of error-checking code your program may require—because all errors filter up to handlers, you may not need to test the outcome of every operation (see this chapter's sidebar “[Why You Will Care: Error Checks](#)” on page 884 for an illustration).
2. Any uncaught exception eventually filters up to the default exception handler Python provides at the top of your program. This handler prints the familiar error message and shuts down your program.
3. If you don't want the default message and shutdown, you can code try statements with except clauses to catch and recover from exceptions that are raised within its nested code block. Once an exception is caught, the exception is terminated and your program continues after the try.
4. The `raise` and `assert` statements can be used to trigger an exception, exactly as if it had been raised by Python itself. In principle, you can also raise an exception by making a programming mistake, but that's not usually an explicit goal!
5. The try statement with a finally clause can be used to ensure actions are run after a block of code exits, regardless of whether the block raises an exception or not. The with statement can also be used to ensure termination actions are run, but only when processing object types that support it.

Why You Will Care: Error Checks

One way to see how exceptions are useful is to compare coding styles in Python and languages without exceptions. For instance, if you want to write robust programs in the C language, you generally have to test return values or status codes after every operation that could possibly go astray, and propagate the results of the tests as your programs run:

```
doStuff()
{
    if (doFirstThing() == ERROR)    # C program
        return ERROR;             # Detect errors everywhere
    if (doNextThing() == ERROR)    # even if not handled here
        return ERROR;
    ...
    return doLastThing();
}

main()
{
    if (doStuff() == ERROR)
        badEnding();
    else
        goodEnding();
}
```

In fact, realistic C programs often have as much code devoted to error detection as to doing actual work. But in Python, you don't have to be so methodical (and neurotic!). You can instead wrap arbitrarily vast pieces of a program in exception handlers and simply write the parts that do the actual work, assuming all is normally well:

```
def doStuff():
    doFirstThing()
    doNextThing()
    ...
    doLastThing()

if __name__ == '__main__':
    try:
        doStuff()
    except:
        badEnding()
    else:
        goodEnding()
```

Because control jumps immediately to a handler when an exception occurs, there's no need to instrument all your code to guard for errors, and there's no extra performance overhead to run all the tests. Moreover, because Python detects errors automatically, your code often doesn't need to check for errors in the first place. The upshot is that exceptions let you largely ignore the unusual cases and avoid error-checking code that can distract from your program's purpose.

Exception Coding Details

The prior chapter provided a quick look at exception-related statements in action. Here, we're going to dig a bit deeper—this chapter provides fuller coverage of exception-processing syntax in Python. Specifically, we'll explore the details behind the `try`, `raise`, `assert`, and `with` statements. Although these statements are mostly straightforward, you'll find that they offer powerful tools for dealing with exceptional conditions in Python code.

The `try` Statement

First up, the `try` statement is how your code catches exceptions. In short, if an exception occurs while running this statement's main block, the program jumps back to run one of the statement's *handlers* and continues from there. Its handlers may be specified by `except`, `else`, `finally`, and `except*` *clauses* nested in the `try`, and separate rules apply to these clauses' syntax, and their valid combinations.

This is a simple model on the surface, but the `try` statement's handler clauses have disjoint purposes, and its rules for valid combinations mean that it comes in distinct flavors. Because of this, we'll approach this subject by exploring the `try`'s common roles in isolation first and putting their pieces together later as a combined statement. This parallels the fact that `try` really *was* separate statements in Python's dim past, but our focus here is on its unified present.

Although technically part of the `try`, we'll also defer the `except*` clause until the next chapter, partly because it encroaches on that chapter's *exception-object* topic, but mostly because this is a tool that complicates the `try` story substantially for an extension that's rarely useful in practice. Our priority here is learning the fundamentals.

`try` Statement Clauses

When you write a `try` statement, a variety of clauses can appear after and below the `try` header. [Table 34-1](#) summarizes all the possible forms as both reference and preview. We've already seen that `except` clauses catch exceptions and `finally` clauses run on the way out. New here, `else` clauses run if no exceptions are encountered, and `except*` clauses process exception groups and support all `except` forms except the empty.

Formally, a `try` must use at least one of the clauses in [Table 34-1](#). There may be any number of `except` clauses, but you can code `else` only if there is at least one `except`, and there can be only one `else` and one `finally`. A `finally` can appear in the same statement as `except` and `else`, with ordering rules given later in this chapter.

Table 34-1. *try* statement clauses and forms

Clause form	Interpretation
<code>except:</code>	Catch all (or all other) exception types
<code>except name:</code>	Catch a specific exception only
<code>except name as var:</code>	Catch the listed exception and assign its instance
<code>except (name1, name2):</code>	Catch any of the exceptions listed in a tuple
<code>except (name1, name2) as var:</code>	Catch any listed exception and assign its instance
<code>else:</code>	Run if no exceptions are raised in the <code>try</code> block
<code>finally:</code>	Always perform this block on exit, exception or not
<code>except* ...nonempty except forms...:</code>	Catch multiple exceptions in a group (Chapter 35)

We'll explore the `as var` part available in some of [Table 34-1](#)'s clauses in more detail when we meet the `raise` statement later in this chapter because it provides access to the object raised as an exception via `var`. Before all that, let's get started by examining the more common clauses of [Table 34-1](#) more closely.

The `except` and `else` Clauses

Syntactically, the `try` is a compound, multipart statement. It starts with a `try` header line, followed by a block of (usually) indented statements, which is followed by clauses that each identify a condition to be handled and give a block of statements to handle it. In its most common form, `try` is coded with one or more `except` clauses and an optional `else` clause at the end. You associate the words `try`, `except`, and `else` by indenting them to the same level (i.e., lining them up vertically), like this:

```
try:
    statements                # Run this main action first
except name1:
    statements                # Run if name1 is raised during the try block
except (name2, name3):
    statements                # Run if name2 or name3 occur in the try
except name4 as var:
    statements                # Run if name4 is raised, and assign it to var
except:
    statements                # Run for all other exceptions raised
else:
    statements                # Run if no exception was raised in the try block
```

Semantically, the block under the `try` header in this statement represents the *main action* of the statement—the code you're trying to run, and wrapping in exception handlers. The rest of the statement defines the handlers themselves: `except` clauses give handlers for exceptions raised during the `try` block, and the optional `else` clause gives a handler run if *no* exceptions occur in the `try` block.

Within a `try`, each `except` names exceptions to catch: a *single* exception catches just that exception, a tuple catches *any* exception in the tuple, and an `except` that omits the exception altogether matches *all* (or all *other*) exceptions. Each nonempty `except` can also give a variable name after `as` to be assigned the exception object raised by Python or `raise` statements; again, we'll explore this option ahead.

How try statements work

Operationally, here's how try statements are run. When a try statement is entered, Python records the current program context so it can return to it if an exception occurs. The statements nested under the try header are run first. What happens next depends on whether an exception is raised while the try block's statements are running, and whether a raised exception matches any of those that the try is watching for:

Exception and match

If an exception occurs while the try block's statements are running, and the exception *matches* one that the statement names, Python jumps back to the try and runs the statements under its topmost except clause that matches the raised exception, after assigning the raised exception object to the variable named by `as` in the clause (if present). After the except block runs, control resumes below the entire try statement. If the except block itself raises another exception, the propagation process is started anew from this point in the code.

Exception and no match

If an exception occurs while the try block's statements are running, but the exception *does not* match one that the statement names, the exception is propagated up to the next most recently entered try statement that matches the exception; if no such matching try statement can be found and the search reaches the top level of the program, Python prints a default error message and terminates the program (unless it's the REPL).

No exception

If an exception does *not* occur while the try block's statements are running, Python runs the statements under the `else` clause (if present), and control then resumes below the entire try statement. If the `else` block itself raises another exception, it kicks off the propagation process again.

In sum, except clauses catch any matching exceptions that happen while the try block is running, and the `else` clause runs only if no exceptions happen while the try block runs. Exceptions raised are *matched* to exceptions named in except clauses by *class* relationships we'll explore both ahead and in the next chapter (brief: a subclass matches its superclass), and the *empty* except clause with no exception name matches any exception.

In effect, except clauses are *focused* exception handlers—they catch exceptions that occur only within the statements in the associated try block. However, as the try block's statements can call functions coded elsewhere in a program, the source of an exception may very well be outside the code of the try statement itself.

In fact, a try block might invoke arbitrarily large amounts of program code—including code that may have try statements of its own, which will be searched first when exceptions occur. In other words, because try statements can nest at runtime, where an exception goes depends on the code run before it, a phenomenon we'll explore in [Chapter 36](#).

If a `finally` clause is added to a try, its code block is run for all three of the cases listed previously, as you'll see ahead. First, though, let's take a look at some common variations of exception-catching clauses.

Catching many exceptions with a tuple

Per the fourth and fifth entries in [Table 34-1](#), except clauses that list *many* exceptions in a parenthesized tuple catch *any* of the listed exceptions. Because Python looks for a match within a given try by

inspecting the `except` clauses from *top to bottom*, the tuple version has the same effect as listing each exception in its own `except` clause, but you have to code the common statement body associated with each only once.

Here’s a partial example of multiple `except` clauses at work, which demos just how specific your handlers can be:

```
try:
    ...
except NameError:
    ...
except IndexError:
    ...
except (AttributeError, TypeError, SyntaxError):
    ...
```

If an exception is raised while this `try` block is running, Python returns to the `try` and searches for the first `except` that names the exception raised. It inspects clauses from top to bottom—and left to right along the way—and runs the statements under the first clause that matches. If none match, the exception is propagated past this `try`.

Note that *parentheses* are required around the tuple in the “any” form, and using an `as` in this form lets you check which exception occurred when you listed many:

```
try:
    ...
except (AttributeError, TypeError, SyntaxError) as What:
    ...and check What...
```

To learn more about both `as`, as well as what happens when no `except` matches, we must move on.

Catching all exceptions with empties and `Exception`

Per the first entry in [Table 34-1](#), `except` clauses that list *no* exception name catch *all* exceptions not previously listed in the `try` statement. That is, if you want to code a general “catchall” handler to be run when no other `except` clause matches the exception raised, an empty `except` does the trick:

```
try:
    ...
except NameError:
    ... # Handle NameError
except IndexError:
    ... # Handle IndexError
except:
    ... # Handle all other exceptions
else:
    ... # Handle the no-exception case (preview)
```

The empty `except` clause is a sort of *wildcard* feature—because it catches everything, adding it to the mix allows your handlers to be as general or specific as you like. In some scenarios, this form may be more convenient than listing all possible exceptions in a `try`—especially when working interactively in a REPL, or writing code that must recover no matter what occurs. For example, the following catches *everything* by not listing anything:

```
try:
    ...
except:
    ... # Catch all possible exceptions
```

That being shown, the empty `except` can also cause problems. It may catch *unexpected* exceptions, and intercept events unrelated to your code and *required* by another handler. For example, even *system exit* calls and Ctrl+C key-combination interrupts in Python work by triggering exceptions, and you usually want these to pass.

Perhaps worse, the empty `except` may also catch genuine *programming mistakes* for which you probably want to see an error message. Otherwise, you may not even know that a bug exists until it's too late to avoid a user's report. We'll revisit this as a gotcha at the end of this part of the book. For now, the standard “use with care” applies.

Python provides an alternative that solves at least one of these problems—catching the built-in `Exception` has almost the same effect as an empty `except`, but won't catch exceptions related to system exits and Ctrl+C:

```
try:
    ...
except Exception:
    ...                # Catch all possible exceptions - except exits
```

We'll explore how this form does its magic formally in the next chapter when we study exception classes. In short, it works because exceptions match if they are a *subclass* of one named in an `except` clause, and `Exception` is a superclass of all the exceptions you should generally catch this way. This form has most of the same convenience of the empty `except` without the risk of catching exit events and also allows you to check the exception raised via `as`. While better, though, it also has some of the same risks—it may still mask and silently ignore programming errors.

The opening snippet of this section deliberately listed an `else` clause, to call out that it is *not* a catchall like the empty `except`—an understandable source of confusion for `try` newcomers. The next section dissects the difference.

Catching the no-exception case with `else`

All told, `else` can be used in *three* places in Python: in `if` selections, `for` and `while` loops, and `try` exception handlers. In the latter, `else` is run when *no* exception occurs—not for *unmatched* exceptions (that's what the prior section's empty `except` is for). This `else` role may seem different than in `if` and loops, but this context differs.

The need for an `else` clause in `try` is not always obvious to Python beginners. Without it, though, there is no direct way to tell whether the flow of control has proceeded past a `try` statement because no exception was raised, or because an exception occurred and was handled. Either way, we wind up after the `try`:

```
try:
    ...run code...
except IndexError:
    ...handle exception...
# Did we get here because the try failed or not?
```

Of course, we could initialize, set, and check a Boolean flag to know what happened, which adds lines of admin code. Much like the way `else` clauses in loops make the exit cause more apparent (for exits sans `break`), the `else` clause provides syntax in `try` that makes the outcome unambiguous with minimal extra code:

```

try:
    ...run code...
except IndexError:
    ...handle exception...
else:
    ...no exception occurred...

```

You can *almost* emulate an `else` clause by moving its code into the `try` block:

```

try:
    ...run code...
    ...no exception occurred...
except IndexError:
    ...handle exception...

```

This can lead to incorrect exception classifications, though. If the “no exception occurred” action itself causes an `IndexError`, it will register as a failure of the `try` block and erroneously trigger the exception handler below the `try` (unlikely perhaps, but true). By using an explicit `else` clause instead, you make the logic more obvious and guarantee that `except` handlers will run only for real failures in the code you’re wrapping in a `try`, not for failures in the `else` no-exception case’s action.

Example: Default behavior

Because the control flow through a program may be easier to capture in Python than in English, let’s run some simple examples that further illustrate exception basics with real code in files.

As noted, exceptions not caught by `try` statements percolate up to the top level of the Python process and run Python’s default exception-handling logic (i.e., Python terminates the running program and prints a standard error message). To illustrate, module file *crashware.py* coded in [Example 34-1](#) generates a divide-by-zero exception—by design.

Example 34-1. *crashware.py*

```

def gobad(x, y):
    return x / y

def gosouth(x):
    print(gobad(x, 0))

if __name__ == '__main__': gosouth(1)

```

Because the program ignores the exception it triggers, Python kills the program and prints a message (edited here to condense paths and drop some interfaces’ code-pointer lines for space):

```

$ python3 crashware.py
Traceback (most recent call last):
  File ".../LP6E/Chapter34/crashware.py", line 7, in <module>
    if __name__ == '__main__': gosouth(1)
  File ".../LP6E/Chapter34/crashware.py", line 5, in gosouth
    print(gobad(x, 0))
  File ".../LP6E/Chapter34/crashware.py", line 2, in gobad
    return x / y
ZeroDivisionError: division by zero

```

This message consists of a stack trace (“Traceback”) and the name of and details about the exception that was raised. The stack trace lists all lines active when the exception occurred, from oldest to newest. Note that because this code was written in a file instead of a REPL, the file and line-number

information is more useful here. For example, we can see that the bad divide happens at the last entry in the trace—line 2 of the file *crashware.py*, a `return` statement.

Because Python detects and reports all errors at runtime by raising exceptions like this, exceptions are intimately bound up with the ideas of *error handling* and *debugging* in general. If you’ve worked through this book’s examples, you’ve undoubtedly seen an exception or two along the way—even typos usually generate a `SyntaxError` or other exception when a file is imported or executed (that’s when the code compiler is run).

By default, programming errors generate a useful error display like the one just shown, which helps you track down the problem. In some interfaces, this message today even comes with pointers to offending expressions, as well as speculative but mandatory “Did you?” tips (whose merit you may wish to reweigh later in your Python career):

```
$ python3
>>> import crashware
>>> crashware.gobad(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../LP6E/Chapter34/crashware.py", line 2, in gobad
    return x / y
          ~^^~
ZeroDivisionError: division by zero
```

Often, this standard error message is all you need to resolve problems in your code. For more heavy-duty debugging jobs, you can catch exceptions with `try` statements, or use debugging tools introduced in [Chapter 3](#), such as the `pdb` standard-library module. See “[Debugging Python Code](#)” on [page 49](#) for more related tips.

Example: Catching built-in exceptions

Python’s error checking and default exception handling is often exactly what you want: especially for code in a top-level script file, an error often *should* terminate your program immediately. For many programs, there is no need to be more specific about errors in your code.

Sometimes, though, you’ll want to catch errors and recover from them instead. If you don’t want your program terminated when Python raises an exception, simply catch it by wrapping the program logic in a `try`. This is an important capability for programs such as network servers, which must keep running persistently. For example, the code in [Example 34-2](#), file *kaboom.py*, catches and recovers from the `Type` Error Python raises immediately when you try to concatenate a list and a string (remember, the `+` operator expects the same sequence type on both sides).

Example 34-2. kaboom.py

```
def kaboom(x, y):
    print(x + y)                                # Trigger TypeError

def serve(n=2):
    for i in range(n):                          # Simulate long-running task
        try:
            kaboom([1, 2], 'hack')
        except TypeError:                       # Catch and recover here
            print('Hello world!')
        print('Resuming here...')              # Continue here if exception or not

if __name__ == '__main__': serve()
```

When the exception occurs in the function `kaboom`, control jumps to the `try` statement's `except` clause, which prints a message. Since an exception is “dead” after it's been caught like this, the program continues executing below the `try` rather than being terminated by Python. In effect, the code processes and clears the error, and your script recovers:

```
$ python3 kaboom.py
Hello world!
Resuming here...
Hello world!
Resuming here...
```

Keep in mind that once you've caught an error, control resumes at the place where you caught it (i.e., after the `try`); there is no direct way to go back to the place where the exception occurred (here, in the function `kaboom`). This makes exceptions more like simple *jumps* than function calls—there is no way to “return” to the scene of the crime.

The finally Clause

The other main flavor of the `try` statement is for coding *finalization* (a.k.a. termination) actions and is really related to exceptions only incidentally. If a `finally` clause is included in a `try`, Python will always run its block of statements “on the way out” of the `try` statement—whether an exception occurred while the `try` block was running or not. That is, this clause doesn't *catch* exceptions, it works around them.

When used in isolation, this flavor's general form is this:

```
try:
    statements                # Run this action first
finally:
    statements                # Always run this code on the way out
```

When a `finally` appears in `try`, Python begins by running the statement block associated with the `try` header line as usual. What happens next depends on whether an exception occurs during the `try` block, and what other clauses are present:

Exception and match

If an exception occurs during the `try` block's run and is matched by an `except` clause, Python first runs the matching `except` block and then runs the `finally` block. After both finish, the program then resumes below the entire `try` statement. The `finally` is also run if the `except` raises a new exception.

Exception and no match

If an exception occurs during the `try` block's run but is *not* caught by an `except`, Python still comes back and runs the `finally` block, but it then propagates the exception up to a previously entered `try` or the top-level default handler. That is, `finally` is run even if an exception is raised and uncaught, but unlike an `except`, the `finally` does not terminate the exception—it continues being raised after the `finally` block runs.

No exception

If an exception does *not* occur while the `try` block is running, Python first runs the `else` block (if present) and then runs the `finally` block. After both finish, the program then resumes below the entire `try` statement. The `finally` is also run if the `else` raises a new exception.

The `try/finally` form is useful when you want to be completely sure that an action will happen after some code runs, regardless of the exception behavior of the code. In practice, it allows you to specify cleanup actions that always must occur, such as file closes and server disconnects where required.

Technically speaking, `finally` can appear in the same statement as `except` and `else`, so there is really a single `try` statement with many optional clauses. Because of its distinct role, though, as well as its ordering rules we will meet in a moment, the `finally` clause may be best thought of as a distinct tool. Whether mixed or not, `finally` serves the same purpose—to specify cleanup actions that must always be run, regardless of any exceptions.

As you'll also see later in this chapter, the `with` statement and its context managers provide an object-based way to do similar work for exit actions. Unlike `finally`, this statement also supports entry actions, but it is limited in scope to objects that implement the context-manager protocol it employs.

Example: Coding termination actions with `try/finally`

We coded some simple `try/finally` examples in the prior chapter. [Example 34-3](#) lists a more tangible example that illustrates a typical role for this statement.

Example 34-3. `closer.py`

```
class MyError(Exception): pass

def stuff(file):
    file.write('Hello?')          # May be delayed in file buffer
    raise MyError()              # <= Enable or disable me with a #

if __name__ == '__main__':
    file = open('temp.txt', 'w')  # Open an output file (this can fail too)
    try:
        stuff(file)              # Raises exception
    finally:
        file.close()             # Always close file to flush output buffers
    print('Am I reached?')        # Continue here only if no exception
```

When the function in this code raises its exception, the control flow jumps back and runs the `finally` block to close the file. The exception is then propagated on to either another `try` or the default top-level handler, which prints the standard error message and shuts down the program. Hence, the statement after this `try` is never reached:

```
$ python3 closer.py
Traceback (most recent call last):
  File ".../LP6E/Chapter34/closer.py", line 10, in <module>
    stuff(file)          # Raises exception
  File ".../LP6E/Chapter34/closer.py", line 5, in stuff
    raise MyError()      # <= Enable or disable me with a #
MyError
```

If the function here did *not* raise an exception (e.g., by disabling its `raise` line with an added `#`), the program would still execute the `finally` block to close the file, but it would then continue below the entire `try` statement:

```
$ python3 closer.py
Am I reached?
```

In this specific case, we’ve wrapped a call to a file-processing function in a `try` with a `finally` clause to make sure that the file is always closed, and thus finalized, whether the function triggers an exception or not. This way, later code can be sure that the file’s output buffer’s content has been flushed from memory to disk. A similar code structure can guarantee that server connections are closed, GUI windows are closed, and so on.

As we learned in [Chapter 9](#), file objects are automatically closed on garbage collection in standard Python (CPython); this is especially useful for temporary files that we don’t assign to variables. However, it’s not always easy to predict when garbage collection will occur, especially in larger programs or alternative Python implementations with differing garbage collection policies. The `try` statement makes file closes more explicit and predictable: it ensures that the file will be closed on block exit, regardless of whether an exception occurs or not.

This particular example’s function isn’t all that useful (it always raises an exception!), but wrapping calls in `try/finally` statements is a good way to ensure that your closing-time termination activities always run. All bets are off if Python itself crashes completely, of course, but this is exceedingly rare; because it detects errors as a program runs, hard crashes are usually caused by linked-in C extension code, outside of Python’s scope.

As a preview, notice how the user-defined exception in [Example 34-3](#) is defined with a *class*; as you’ll learn more formally in the next chapter, exceptions must all be class instances, for reasonably good causes.

Combined try Clauses

For the first 15 years of Python’s tenure (more or less), the `try` statement came in two flavors and was two separate statements—we could either use a `finally` to ensure that cleanup code was always run, or write `except` blocks to catch and recover from specific exceptions and optionally specify an `else` clause for when exceptions occurred.

That is, the `finally` clause could not be *mixed* with `except` and `else`. This was partly because of implementation issues, and partly because the meaning of mixing the two seemed obscure—catching and recovering from exceptions seemed a disjoint concept from performing cleanup actions.

For better or worse, the two statements eventually merged. Today, we can mix `finally`, `except`, and `else` clauses in the same statement—in part because of similar utility in the Java language (alas, many a programming-language mod owes to imitation). That is, the `try` statement in its most complete form looks like this:

```
try:                                # Combined try statement
    main-action
except Exception1:                # Catch specific exceptions
    handler1
except Exception2:
    handler2
except:                             # Catch all (other) exceptions
    handler3
else:                               # No-exception handler
    handler4
finally:                            # The finally encloses all
    finally-block
```

The code in this statement’s *main-action* block is executed first, as usual. If that code raises an exception, all the `except` blocks are tested, one after another, for a match to the exception raised: *handler1* is

run for *Exception1*, *handler2* for *Exception2*, and *handler3* for all others. If no exception is raised, *handler4* is run.

No matter what's happened previously, the *finally-block* is executed once, after the main action block is exited and any handler block has been run. In fact, the code in the *finally-block* will be run even if an error or `raise` in an `except` or `else` block causes a new exception to be raised.

As outlined earlier, even in mixed usage like this, the `finally` clause does not end the exception—if an exception is active when the *finally-block* is executed, it continues to be propagated after the *finally-block* runs, and control jumps somewhere else in the program (to an earlier `try`, or to the default top-level handler). If no exception is active when the `finally` is run, control resumes after the entire `try` statement.

The net effect is that the `finally` is always run, regardless of whether:

- An exception occurred in the main action and was handled.
- An exception occurred in the main action and was not handled.
- No exceptions occurred in the main action.
- A new exception was triggered in one of the handlers.

Again, the `finally` serves to specify cleanup actions that must always occur on the way out of the `try`, regardless of what exceptions have been raised or handled.

Combined-clause syntax rules

When combined like this, the `try` statement must have either an `except` or a `finally`, and the order of its parts must be like this (where “->” means “is followed by”):

```
try -> except -> else -> finally
```

In this, the `else` and `finally` are optional, and there may be zero or more `excepts`, but there must be at least one `except` if an `else` appears. Really, the `try` statement consists of two parts: `excepts` with an optional `else`, and/or the `finally`.

In fact, it's more accurate to describe the combined `try` statement's syntactic by the following two alternative formats (where square brackets mean optional and star means any number of what precedes it):

```
try:                                # Format 1
    statements
except [type [as value]]:
    statements
[except [type [as value]]:
    statements]*
[else:
    statements]
[finally:
    statements]

try:                                # Format 2
    statements
finally:
    statements
```

Because of these rules, the `else` can appear only if there is at least one `except`, and it's always possible to mix `except` and `finally`, regardless of whether an `else` appears or not. It's also possible to mix `finally`

and else, but only if an except appears too (though the except can omit an exception name to catch everything and run a raise statement, described later, to reraise the current exception). If you violate any of these (arguably intricate!) ordering rules, Python will raise a syntax error exception before your code runs.

Combining finally and except by nesting

It may help to realize that it's also possible to combine finally and except clauses in a try by syntactically nesting a try/except in the try block of a try/finally statement. We'll explore this technique more fully in [Chapter 36](#), but the following has the same effect as the combined form shown at the start of this section:

```
try:                                     # Nested equivalent to combined form
    try:
        main-action
    except Exception1:
        handler1
    except Exception2:
        handler2
    except:
        handler3
    else:
        handler4
finally:
    finally-block
```

Again, the finally block is always run on the way out, regardless of what happened in the main action and regardless of any exception handlers run in the nested try (trace through the four cases listed previously to see how this works the same). Since an else always requires an except, this nested form even sports the same mixing constraints of the combined form outlined in the preceding section.

However, this nested equivalent seems more obscure to some people and requires more code than the new merged form—though just one four-character line plus extra indentation. Mixing finally into the same statement might make your code easier to write and read, though this also might depend on who you ask.

Combined-clauses example

To demo the effect of mixing finally with other try clauses, the script listed in [Example 34-4](#), *trycombos.py*, codes four common scenarios, with print statements that describe the meaning of each.

Example 34-4. trycombos.py

```
sep = '-' * 45 + '\n'

print(sep + 'EXCEPTION RAISED AND CAUGHT')
try:
    x = 'hack'[99]
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')
```

```

print(sep + 'EXCEPTION NOT RAISED')
try:
    x = 'hack'[3]
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')

print(sep + 'EXCEPTION NOT RAISED, WITH ELSE')
try:
    x = 'hack'[3]
except IndexError:
    print('except run')
else:
    print('else run')
finally:
    print('finally run')
print('after run')

print(sep + 'EXCEPTION RAISED BUT NOT CAUGHT')
try:
    x = 1 / 0
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')

```

When this code is run, the following output is produced. Trace through the code to see how exception handling produces the output of each of the four tests here:

```

$ python3 trycombos.py
-----
EXCEPTION RAISED AND CAUGHT
except run
finally run
after run
-----
EXCEPTION NOT RAISED
finally run
after run
-----
EXCEPTION NOT RAISED, WITH ELSE
else run
finally run
after run
-----
EXCEPTION RAISED BUT NOT CAUGHT
finally run
Traceback (most recent call last):
  File ".../LP6E/Chapter34/trycombos.py", line 38, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero

```

This example uses built-in operations in the main action to trigger exceptions (or not), and it relies on the fact that Python always checks for errors as code is running. The next section shows how to raise exceptions manually instead.

The raise Statement

To trigger exceptions explicitly, code `raise` statements. Their general form is simple—a `raise` statement consists of the word `raise`, optionally followed by the class to be raised, or an instance of it:

```
raise instance      # Raise an instance of a class
raise class         # Make and raise an instance of a class
raise               # Reraise the most recent exception
```

As mentioned earlier, exceptions are always instances of *classes* today. Hence, the first `raise` form here is the most common—we provide an *instance* directly, either created before the `raise` or within the `raise` statement itself. If we pass a *class* instead, Python calls the class with *no* constructor arguments, to create an instance to be raised; this form is equivalent to adding parentheses after the class reference. The last form reraises the most recently raised exception; it's commonly used in exception handlers to propagate exceptions that have been caught.



Blast from the past: Long ago and far away (well, before Python 2.6), exceptions could be identified as simple *string* objects, with an optional associated data item in `raise`. This was replaced with *classes* to support added functionality and categories, as you'll see in the next chapter. Still, strings were a simpler model for simpler roles, didn't require newcomers to learn classes and OOP before exceptions, and didn't force some Python books to put exceptions on hold until **Part VII!**

Raising Exceptions

To make `raise` more concrete, let's turn to some examples. With built-in exceptions, the following two forms are equivalent—both raise an instance of the exception class named, but the first creates the instance implicitly:

```
raise IndexError      # Class (instance created)
raise IndexError()    # Instance (created in statement)
```

We can also create the instance ahead of time—because the `raise` statement accepts any kind of object reference, the following two examples raise `IndexError` just like the prior two:

```
exc = IndexError()    # Create instance ahead of time
raise exc

excs = [IndexError, TypeError]
raise excs[0]
```

In fact, the instance provided to `raise` can be had in the `try` that catches it too, per the next section.

The `except` as hook

When an exception is raised, Python sends the raised instance along with the exception. If a `try` includes an `except` with an `as` clause per **Table 34-1**, the variable it gives will be assigned the instance raised by `raise` or Python:

```
try:
    ...
except IndexError as X:    # X assigned the raised instance object
    ...
```

The `as` is optional in a `try` handler (if it's omitted, the instance is simply not assigned to a name), but including it allows the handler to access both data in the instance and methods in the exception class.

This model works the same for user-defined exceptions we code with classes—the following, for example, passes to the exception class constructor arguments that become available in the handler through the assigned instance:

```
class MyExc(Exception): pass

try:
    raise MyExc('oops')      # Exception class with constructor args
except MyExc as X:          # Instance attributes available in handler
    print(X.args)            # Prints ('oops',)
```

Because this encroaches on the next chapter's topic, though, we'll defer further details until then.

Regardless of their source, exceptions are always identified by class *instance* objects, and at most one is active at any given time (sans the `except*` groups of [Chapter 35](#)). Once caught by an `except` clause anywhere in the program, an exception ends and won't propagate to another `try`, unless it's reraised by another `raise` statement or error.

Scopes and `except as`

We'll study exception objects in more detail in the next chapter. Now that we've seen the `as` variable in action, though, we can finally clarify the related scope issue summarized back in [Chapter 17](#). As mentioned in that chapter, the variable used to access an exception in the `as` clause of an `except` is localized to the `except` block—the variable is not available after the block exits, much like a temporary loop variable in comprehension expressions:

```
$ python3
>>> try:
...     1 / 0
... except Exception as X:          # The "as" localizes names to except block
...     print(X)
...
division by zero
>>> X
NameError: name 'X' is not defined
```

Unlike comprehension loop variables, though, this variable is *removed* after the `except` block exits. This is done because the variable would otherwise retain a reference to the runtime call stack, which would defer garbage collection and thus retain excess memory space. This removal occurs, though, *even* if you're using the name for other purposes in the surrounding scope, and is a much more extreme policy than that used for comprehensions:

```
>>> X = 99
>>> {X for X in 'hack'}          # Comprehensions localize but don't remove
{'a', 'c', 'k', 'h'}
>>> X
99

>>> X = 99
>>> try:
...     1 / 0
... except Exception as X:      # But "as" localizes _and_ removes on exit!
...     print(X)
...
division by zero
```

```
>>> X                                     # Where did my X go? - an odd boundary case
NameError: name 'X' is not defined
```

Because of this, you should generally use unique variable names in your `try` statement's `except` clauses, even if they are localized by scope. If you do need to reference the exception instance after the `try` statement, simply assign it to another name that won't be automatically removed:

```
>>> try:
...     1 / 0
... except Exception as X:                 # Python removes this reference
...     print(X)
...     saveit = X                         # Assign exc to retain exc if needed
...
division by zero
>>> X
NameError: name 'X' is not defined
>>> saveit
ZeroDivisionError('division by zero',)
```

Propagating Exceptions with `raise`

The `raise` statement is a bit more feature-rich than we've seen thus far. For example, a `raise` that does not list an exception to raise simply reraises the currently active exception. This form is typically used if you need to catch and handle an exception but don't want the exception to die in your handler:

```
>>> try:
...     raise IndexError('code')           # Exceptions remember arguments
... except IndexError:
...     print('propagating')
...     raise                             # Reraise most recent exception
...
propagating
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: code
```

Running a `raise` this way reraises the exception and propagates it to a higher handler (or the default handler at the top, which stops the program with a standard error message). Notice how the argument we passed to the exception class shows up in the error messages; you'll learn why this happens in the next chapter.

Exception Chaining: `raise from`

Exceptions can sometimes be triggered in response to other exceptions—both deliberately and by new program errors. To support full disclosure in such cases, Python also allows `raise` statements to have an optional `from` clause:

```
raise newexception from otherexception
```

When the `from` is used in an *explicit* `raise` request, the expression following `from` specifies another exception class or instance to attach to the `__cause__` attribute of the new exception being raised. If the raised exception is not caught, Python prints both exceptions as part of the standard error message:

```
>>> try:
...     1 / 0
... except Exception as E:
...     raise TypeError('Bad') from E     # Explicitly chained exceptions
...
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: Bad
```

When an exception is raised *implicitly* by a program error inside an exception handler, a similar procedure is followed automatically: the previous exception is attached to the new exception's `__context__` attribute and is again displayed in the standard error message if the exception goes uncaught:

```
>>> try:
...     1 / 0
... except:
...     badname                                # Implicitly chained exceptions
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'badname' is not defined
```

In both cases, because the original exception objects thus attached to new exception objects may *themselves* have attached causes, the causality chain can be *arbitrarily long*, and is displayed in full in error messages. That is, error messages might give more than two exceptions. The net effect in both explicit and implicit chaining contexts is to allow programmers to know all exceptions involved when one exception triggers another:

```
>>> try:
...     try:
...         raise IndexError()
...     except Exception as E:
...         raise TypeError() from E
... except Exception as E:
...     raise SyntaxError() from E
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
IndexError
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
TypeError
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
SyntaxError: None
```

Code like the following similarly displays three exceptions, though implicitly triggered by handler errors (its separator lines are “During handling of the above exception...” instead of “The above exception was the direct cause...”):

```
try:
    try:
        1 / 0
    except:
        badname
except:
    open('nonesuch')
```

Exception chains impact error displays, but do not affect the way that exceptions are named and caught in `try` statements: chains are simply recorded in exception-object attributes which may be inspected as usual where useful.

Like the combined `try`, chained exceptions are similar to utility in other languages (including Java and C#) though it's not clear which languages were borrowers. In Python, it's not unusual to see exception chains in error messages, but it is uncommon to create them explicitly with `raise`, so we'll defer to Python's manuals for more details.

As a footnote on this topic, though, Python also provides a way to *stop* exceptions from chaining: a `raise from None` allows the display of the chained exception context to be disabled when needed. This makes for less cluttered error messages in applications that convert between exception types while processing exception chains.

The `assert` Statement

As a somewhat special case for debugging purposes, Python also includes the `assert` statement in its exceptions toolset. It is mostly just syntactic shorthand for a common `raise` usage pattern, and an `assert` can be thought of as a *conditional* `raise` statement. A statement of the form:

```
assert test, data           # The data part is optional
```

works like the following code:

```
if __debug__:
    if not test:
        raise AssertionError(data)
```

In other words, if the *test* evaluates to false, Python raises an exception: the *data* item (if it's provided) is used as the exception's constructor argument. Like all exceptions, the built-in `AssertionError` exception will kill your program if it's not caught with a `try`, and the *data* item shows up as part of the standard error message:

```
>>> language = 'Java'
>>> assert language.startswith('Py'), "You're using the wrong language!"
AssertionError: You're using the wrong language!
```

As an added feature, `assert` statements are removed from a compiled program's bytecode—and hence not run—if the `-O` Python command-line flag is used to optimize the program. The `__debug__` flag is a built-in and unchangeable name that is automatically set to `True` unless the `-O` flag is used. When `__debug__` is `False` for `-O`, any code predicated on it being `True` is removed, including asserts.

Hence, to disable (and omit) asserts, run code with a command line like `python -O file.py`, or generate optimized bytecode before program runs with similar options in the `compileall` standard-library module or `compile` built-in function. See Python's manuals for details on the module and function.

Example: Trapping Constraints (but Not Errors!)

Here's a less politically charged example of `assert` in action. Assertions are typically used to verify program conditions during development. When displayed, their error message text automatically includes source code line information and the value listed in the `assert` statement. Consider the file *asserter.py* in [Example 34-5](#).

Example 34-5. asserter.py

```
def f(x):
    assert x < 0, 'x must be negative'
    return x ** 2
```

Running this normally triggers the assertion error for positive numbers, but running with `-O` does not:

```
$ python3
>>> import asserter
>>> asserter.f(-3)
9
>>> asserter.f(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    asserter.f(3)
  File ".../LP6E/Chapter34/asserter.py", line 2, in f
    assert x < 0, 'x must be negative'
AssertionError: x must be negative

$ python3 -O
>>> import asserter
>>> asserter.f(3)
9
```

It's important to keep in mind that `assert` is mostly intended for trapping user-defined constraints, not for catching genuine programming *errors*. Because Python traps programming errors itself, there is usually no need to code `assert` to catch things like out-of-bounds indexes, type mismatches, and zero divides:

```
def reciprocal(x):
    assert x != 0          # A generally useless assert!
    return 1 / x          # Python checks for zero automatically
```

Such `assert` use cases are usually superfluous—because Python raises exceptions on errors automatically, you might as well let it do the job for you. As a rule, you normally don't need to do error checking explicitly in your own code.

Of course, there are exceptions to most rules. As suggested earlier in the book, if a function has to perform long-running or unrecoverable actions before it reaches the place where an exception will be triggered, you still might want to test for errors. Even in this case, though, be careful not to make your tests overly specific or restrictive, or you will limit your code's utility.

For another example of common `assert` usage, see the abstract superclass example in [Chapter 29](#); there, we used `assert` to make calls to undefined methods fail with a message. It's a rare but useful tool.

The with Statement and Context Managers

In addition to the tools we’ve seen so far, Python includes another that delegates exception-related tasks to objects. The `with` statement is designed to work with *context manager* objects that support a method-based protocol. The combination is similar in spirit to the way that iteration tools like `for` work with methods of the iteration protocol.

The `with` statement is also similar to a “using” statement in the C# language. Although a somewhat optional and advanced tools-oriented topic (and once a candidate for the next part of this book), context managers are lightweight and useful enough to group with the rest of the exception toolset here.

In short, the `with` statement is designed to be an alternative to a common `try/finally` usage idiom: like that statement, `with` is in large part intended for specifying termination-time or “cleanup” activities that must run regardless of whether an exception occurs during the execution of a block of code.

Unlike `try/finally`, the `with` statement is based upon a method-call protocol for specifying actions to be run around a block of code. This makes `with` less general, qualifies it as redundant in termination roles, and requires coding classes for objects that do not support its protocol. On the other hand, `with` also handles entry actions, can reduce code size where supported, and allows code contexts to be managed with full OOP.

Python enhances some built-in tools with context managers, such as files that automatically close themselves, thread locks that automatically lock and unlock, and `async-function` tools that automatically await results per [Chapter 20](#), but programmers can code context managers of their own with classes, too. Let’s take a brief look at the statement and its implicit protocol.

Basic with Usage

The basic format of the `with` statement looks like this, with an optional part in square brackets here:

```
with expression [as variable]:  
    with-block
```

The statements of the nested *with-block* are the main action to be run here. The *expression* is assumed to return an object that supports the context-management protocol (more on this protocol in a moment). This object may also return a value that will be assigned to the name *variable* if the optional `as` clause is present.

Note that the *variable* is not necessarily assigned the result of the *expression*; the result of the *expression* is the object that supports the context protocol, and the *variable* may be assigned something else intended to be used inside the *with-block*. The object returned by the *expression* may then run startup code before the block is started, as well as termination code after the block is done—whether the block raised an exception or not.

As noted, some built-in Python objects have been augmented to support the context-management protocol, and so can be used with the `with` statement. For example, file objects (covered in [Chapter 9](#)) have a context manager that automatically closes the file after the `with` block regardless of whether an exception is raised, and regardless of if or when the version of Python running the code may close automatically. In abstract code:

```
with open('somefile.txt') as myfile:  
    for line in myfile:  
        print(line)
```

Here, the call to `open` returns a simple file object that is assigned to the name `myfile`. We can use `myfile` with the usual file tools—in this case, the file iterator reads line by line in the `for` loop.

However, the `open` result also supports the context-management protocol used by the `with` statement. After this `with` statement has run, the context management machinery guarantees that the file object referenced by `myfile` is automatically closed, even if the `for` loop raised an exception while processing the file.

Although file objects may be automatically closed on garbage collection, it's not always straightforward to know when that will occur, especially when using alternative Python implementations. The `with` statement in this role is an alternative that allows us to be sure that the close will occur automatically after execution of a specific block of code.

As covered earlier, we can achieve a similar effect with the more general and explicit `try/finally` idiom, but it requires three more lines of administrative code in this case (four instead of just one):

```
myfile = open('somefile.txt')
try:
    for line in myfile:
        print(line)
finally:
    myfile.close()
```

Of course, we could skip *both* statements, but our file may not be closed if an exception is raised during the `for` loop, and this can matter in long-running programs (we'll revisit such trade-offs in [Chapter 36](#)).

As another example, we won't cover Python's multithreading modules in this book, but the lock and condition synchronization objects they define may also be used with the `with` statement because they support the context-management protocol—in this case adding both entry and exit actions around a block. After importing `threading`:

```
lock = threading.Lock()
with lock:
    ...access shared resources...
```

Here, the context management machinery guarantees that the lock is automatically *acquired* before the block is executed and *released* once the block is complete, regardless of exception outcomes.

Finally, the `decimal` module introduced in [Chapter 5](#) also uses context managers to simplify saving and restoring the current decimal context, which specifies the precision and rounding characteristics for calculations:

```
with decimal.localcontext() as ctx:
    ctx.prec = 2
    x = decimal.Decimal('1.00') / decimal.Decimal('3.00')
```

After this statement runs, the current thread's context manager state is automatically restored to what it was before the statement began. To do the same with a `try/finally`, we would need to save the context before and restore it manually after the nested block.

The Context-Management Protocol

Although some built-in types come with context managers, we can also write new ones of our own. To implement context managers, classes use special methods that fall into the operator-overloading category to tap into the `with` statement. The interface expected of objects used in `with` statements is somewhat complex, and most programmers only need to know how to use existing context managers. For

tool builders who might want to write new application-specific context managers, though, let's take a quick look at what's involved.

Here's how the `with` statement actually works:

1. The expression is evaluated, resulting in an object known as a *context manager* that must have `__enter__` and `__exit__` methods.
2. The context manager's `__enter__` method is called. The value it returns is assigned to the variable in the `as` clause if present, or simply discarded otherwise.
3. The code in the nested `with` block is executed.
4. If the `with` block raises an exception, the context manager's `__exit__(type, value, traceback)` method is called with the exception details. These are the same three values returned by `sys.exc_info`, described in the Python manuals and later in this part of the book. If this method returns a false value, the exception is reraised; otherwise, the exception is terminated. The exception should normally be reraised so that it is propagated outside the `with` statement after `__exit__` returns.
5. If the `with` block does *not* raise an exception, the `__exit__` method is still called, but its `type`, `value`, and `traceback` arguments are all passed in as `None`, and its return value is ignored.

Let's look at a quick demo of the protocol in action. The file `withas.py` in [Example 34-6](#) defines a context-manager object that simply traces the entry and exit of the `with` block in any `with` statement it is used for.

Example 34-6. withas.py

"A context manager that traces entry and exit of any `with` statement's block"

```
class TraceBlock:
    def message(self, arg):
        print('running ' + arg)

    def __enter__(self):
        print('[starting with block]')
        return self

    def __exit__(self, exc_type, exc_value, exc_tb):
        if exc_type is None:
            print('[exited normally]\n')
        else:
            print(f'[propagating exception: {exc_type}]')
            return False

if __name__ == '__main__':
    with TraceBlock() as action:
        action.message('test 1')
        print('reached')

    with TraceBlock() as action:
        action.message('test 2')
        raise TypeError
    print('not reached')
```

Notice that this class's `__exit__` method returns `False` to propagate the exception; deleting the `return` statement would have the same effect, as the default `None` return value of functions is `false` by definition, but explicit is generally better in coding. Also notice that the `__enter__` method returns `self` as the object to assign to the `as` variable; in other use cases, this might return a completely different object instead.

When run, this module's self-test code uses its context manager to trace the entry and exit of two `with` statement blocks. The net effect automatically invokes the manager's `__enter__` and `__exit__` methods:

```
$ python3 withas.py
[starting with block]
running test 1
reached
[exited normally]

[starting with block]
running test 2
[propagating exception: <class 'TypeError'>]
Traceback (most recent call last):
  File ".../LP6E/Chapter34/withas.py", line 25, in <module>
    raise TypeError
TypeError
```

Context managers can also utilize OOP state information and inheritance, but are somewhat advanced devices meant for tool builders, so we'll skip additional details here. See Python's standard manuals for the full story—including its coverage of the `contextlib` standard module that provides additional tools for coding context managers.

Also remember that the `try/finally` combination provides support for termination-time activities too, and is generally sufficient in roles that don't warrant coding classes to support the `with` statement's protocol.

Multiple Context Managers

The `with` statement has one last card to turn over: it may also specify *multiple* (sometimes called “nested”) context managers with comma syntax. For example, in the following code snippet that selects lines by substring, both files' exit actions are automatically run to close the files when the statement block exits, regardless of exception outcomes:

```
with open('lines.txt') as input, open('matches.txt', 'w') as output:
    for line in input:
        if 'somekey' in line:
            output.write(line)
```

Any number of context manager items may be listed, and multiple items work the same as nested `with` statements. That is, the following hypothetical code:

```
with A() as a, B() as b:
    statements
```

is equivalent to (and possibly simpler than) the following:

```
with A() as a:
    with B() as b:
        statements
```

The net effect is that each context manager's entry and exit method is run in turn on block entry and exit, and exceptions in the block are caught automatically and possibly reraised on `with` exit at the

discretion of the outermost manager’s exit method. Multiple *file* context managers, for instance, will all be run to open files on entry, and close them on exit—exception or not.

The Termination-Handlers Shoot-Out

You can find more info on context managers in Python’s docs. Rather than getting more detailed here, let’s close out this chapter with a quick look at this extension in action and a vetting of its roles. Using newer and redundant tools like `with` doesn’t in and of itself prove intelligence, and it’s important to understand the trade-offs such options imply.

First up, the following codes a parallel *lines scan* of files located in this book’s examples package. It uses `with` to open two files at once and then reads and zips together their next-line pairs on each iteration of a `for` loop. Thanks to the file object’s context manager, there’s no need to manually catch exceptions or close files when finished:

```
>>> with open('lines1.txt') as file1, open('lines2.txt') as file2:
    for pair in zip(file1, file2):
        print(pair)

('hack\n', 'HACK\n')
('code\n', 'GOOD\n')
('well\n', 'CODE\n')
```

You might also use this coding structure to do a *lines comparison* of two text files. The following simply replaces the former’s `print` with an `if` for a comparison operation, and adds an `enumerate` for automatic line numbers:

```
>>> with open('lines1.txt') as file1, open('lines2.txt') as file2:
    for (linenum, (line1, line2)) in enumerate(zip(file1, file2)):
        if line1.lower() != line2.lower():
            print(f'{linenum} => {line1!r} != {line2!r}')

1 => 'code\n' != 'GOOD\n'
2 => 'well\n' != 'CODE\n'
```

That said, `with` isn’t all that useful in the preceding examples when using CPython, because *input* file objects don’t require a buffer flush, and file objects are *closed* automatically when garbage collected if still open. Moreover, *exceptions* in the `with` block are still propagated outside the statement if not explicitly caught. Hence, the temporary files would be auto-closed immediately and exception behavior would be the same for simpler code like this:

```
for pair in zip(open('lines1.txt'), open('lines2.txt')):    # Same if auto close
    print(pair)                                           # Ditto for != test
```

On the other hand, some of the alternative *Pythons* of [Chapter 2](#) may use different garbage collectors that require direct closes, to avoid taxing system resources. In addition, *output* files may require closes to ensure that any buffered content is transferred to disk so it’s available for opens in later code.

The following *lines filter* code addresses both concerns, by automatically closing files on statement exit, exception or not (it also uses parentheses and line splits after `with`, available as of Python 3.10, and omits write counts for brevity):

```
>>> with (open('lines1.txt') as input,
        open('uppers.txt', 'w') as output):
    for line in input:
        output.write(line.upper())

>>> print(open('uppers.txt').read())      # File content is available here
HACK
CODE
WELL
```

Still, in simple scripts, we can often just open files in separate statements and close after processing if needed. There's no point in catching an exception if it means your program is out of business anyhow, and closes are required to flush output buffers only if files will be reopened by later code—which may never be reached after exceptions anyhow:

```
input = open('lines1.txt')
output = open('uppers.txt', 'w')
for line in input:
    output.write(line.upper())           # Same effect if files auto close,
                                         # and file is not reopened ahead
```

Nevertheless, in programs that must *both* continue after exceptions and close output files for later use in the same program (or REPL) run, the `with` avoids an equivalent `try/finally` combination that may be more obvious to some readers, but also requires noticeably more code—eight lines instead of four, quantitatively speaking:

```
input = open('lines1.txt')
output = open('uppers.txt', 'w')
try:
    for line in input:
        output.write(line.upper())
finally:
    input.close()                       # Ensure output file is complete
    output.close()                     # Whether exception occurs or not
```

Even so, the `try/finally` is a single tool that applies to all finalization cases and makes code explicit. The `with` can be more concise for context-manager users, but applies only to objects that implement its complex protocol, relies on implicit “magic” that obscures meaning, and adds redundancy that doubles the required knowledge base of programmers. As usual, you’ll have to weigh these tools’ trade-offs for yourself.

Chapter Summary

In this chapter, we took a more detailed look at exception processing by exploring the statements related to exceptions in Python: `try` to catch them, `raise` to trigger them, `assert` to raise them conditionally, and `with` to wrap code blocks in context managers that automate entry and exit actions.

Up to this point, exceptions may seem like a fairly lightweight tool (apart from the `with` protocol, that is). The most complex thing about them may be how they are identified—a topic the next chapter will address by showing how exception objects are made. As you’ll see there, classes allow you to code new exceptions specific to your programs. Before we move ahead, though, let’s work through the following short quiz on the basics covered here.

Test Your Knowledge: Quiz

1. What is the `try` statement for?
2. What are the two common variations of the `try` statement?
3. What is the `raise` statement for?
4. What is the `assert` statement designed to do, and what other statement is it like?
5. What is the `with` statement designed to do, and what other statement is it like?

Test Your Knowledge: Answers

1. The `try` statement catches and recovers from exceptions—it specifies a block of code to run and one or more handlers for exceptions that may be raised during the block's execution.
2. The two common variations on the `try` statement are `try/except/else` (for catching exceptions) and `try/finally` (for specifying cleanup actions that must occur whether an exception is raised or not). Despite these logically distinct roles, the `except` and `finally` blocks may be mixed in the same statement, so the two forms are really part of the single `try` statement. Even when mixed with `except`, though, the `finally` is still run on the way out of the `try`, regardless of what exceptions may have been raised or handled. In fact, the combined form is equivalent to nesting a `try/except/else` in a `try/finally`.
3. The `raise` statement raises (triggers) an exception. Python raises built-in exceptions on errors internally, but your scripts can trigger built-in or user-defined exceptions too with `raise`.
4. The `assert` statement raises an `AssertionError` exception if a condition is false. It's similar to a conditional `raise` statement wrapped up in an `if` statement, and can be disabled with a `-O` command switch.
5. The `with` statement is designed to automate startup and termination activities that must occur around a block of code. It is roughly like a `try/finally` combination in that its exit actions run whether an exception occurred or not, but it employs an object-based protocol for specifying entry and exit actions and may reduce code size for context-manger users. Still, it's not quite as general, as it applies only to objects that support its protocol; `try` with `finally` clauses can handle more use cases.

Exception Objects

So far, this book has been somewhat vague about what an exception actually *is*. This chapter clears up the mystery by disclosing the facts behind exception objects—both built-in and user-defined. As suggested in the preceding chapters, exceptions are identified by *class instance objects*. This is what is raised and propagated along by exception processing, and the source of the class matched against `except` clauses in `try` statements.

Although this means you must use object-oriented programming to define new exceptions in your programs—and introduces a knowledge dependency lamented in the prior chapter’s note—basing exceptions on classes and OOP offers a number of benefits. Among them, class-based exceptions support:

Flexible exception categories

Exception classes allow code to choose specificity and ease future changes. Adding new exception subclasses, for example, need not require changes in `try` statements.

State information and behavior

Exception classes provide a natural place to store context for use in the `try` handler. Both attributes and methods, for example, are available on the raised instance.

Reuse by inheritance

Exceptions classes can participate in inheritance hierarchies to obtain and customize common behavior. Inherited error displays, for example, can provide a common look and feel.

Because of these advantages, class-based exceptions support program evolution and larger systems well. As you’ll learn here, all built-in exceptions are identified by classes and are organized into an inheritance tree for the reasons just listed. You can do the same with user-defined exceptions of your own.

In fact, the built-in exceptions we’ll study here turn out to be integral to new exceptions you define. Because Python largely requires user-defined exceptions to inherit from built-in exception classes that provide useful defaults for printing and state, the task of coding user-defined exceptions also involves understanding the roles of these built-ins.

Exception Classes

Whether built-in or user-defined, exceptions work much of their magic by *superclass relationships*: a raised exception matches an `except` clause if that clause names the exception's class or any superclass of it. Put another way, a `try` statement's `except` matches both the class it lists, as well as all of that class's subclasses lower in the class tree.

The net effect is that class exceptions naturally support the construction of exception *hierarchies*: superclasses become *category* names, and subclasses become *specific* kinds of exceptions within a category. By naming a general exception superclass, an `except` clause can catch an entire category of exceptions—any more specific subclass will match.

In addition to this category idea, class-based exceptions support exception state information and allow exceptions to inherit common behaviors, as noted. To see how all these assets come together in code, let's turn to an example.

Coding Exceptions Classes

In the file listed in [Example 35-1](#), *categoric.py*, we define a superclass called `General` and two subclasses called `Specific1` and `Specific2`. This example illustrates the notion of exception categories—`General` is a category name, and its two subclasses are specific types of exceptions within the category. Handlers that catch `General` will also catch any subclasses of it, including `Specific1` and `Specific2`.

Example 35-1. categoric.py

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0():
    X = General()           # Raise superclass instance
    raise X

def raiser1():
    X = Specific1()         # Raise subclass instance
    raise X

def raiser2():
    X = Specific2()         # Raise different subclass instance
    raise X

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General:         # Match General or any subclass of it
        import sys
        print('caught:', sys.exc_info()[0])
```

When this example runs, its `try` statement catches and reports instances of all three of its classes because the `except` clause names their common superclass:

```
$ python3 categoric.py
caught: <class '__main__.General'>
caught: <class '__main__.Specific1'>
caught: <class '__main__.Specific2'>
```

This code is mostly straightforward, but here are a few points to notice:

Exception superclass

Classes used to build exception category trees have very few requirements—in fact, in this example, they are mostly empty, with bodies that do nothing but `pass`. Notice, though, how the top-level class here inherits from the built-in `Exception` class. This is required: classes that don't inherit from a built-in exception class won't work in most exception contexts. The built-in superclass is normally `Exception`, the root for nonexit exceptions, but may also be `BaseException`, the root for all exceptions, or other. Although we don't employ it here, `Exception` provides behavior you'll meet later that makes inheriting from it useful, required or not.

Raising instances

In this code, we call classes to make *instances* for the `raise` statements (notice the parentheses). In the class exception model, we always raise and catch a class instance object. If we list a class name without parentheses in a `raise`, Python makes an instance for us by calling the class with no constructor arguments. Exception instances can be created before the `raise`, as done here, or within the `raise` statement itself.

Catching categories

This code includes functions that raise instances of all three of our classes as exceptions, as well as a top-level `try` that calls the functions and catches `General` exceptions. The same `try` also catches the two specific exceptions because they are subclasses of `General`—that is, members of its category.

Exception details

The exception handler here uses the `sys.exc_info` call, which is one way to fetch the exception being handled in a generic fashion. As you'll see in more detail in the next chapter, the first item in this call's result tuple is the class of the exception raised, and the second is the actual instance raised. In a general `except` clause like the one here that catches all classes in a category, `sys.exc_info` can be used to determine exactly what has occurred.

The last point merits elaboration. When an exception is caught, we can be sure that the instance raised is an instance of the class listed in the `except` or one of its subclasses. Because of that, the specific kind of exception raised can also be had via the `type` result or `__class__` attribute of the instance, regardless of how the instance is obtained.

To demo, the variant in [Example 35-2](#) works the same as the prior example but uses the `as` extension in its `except` clause to directly assign a variable to the instance raised, from which `type` yields the exception's kind.

Example 35-2. `categoric2.py`

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0(): raise General()
def raiser1(): raise Specific1()
def raiser2(): raise Specific2()

for func in (raiser0, raiser1, raiser2):
    try:
        func()
```

```
except General as X:                # X is the raised instance
    print('caught:', type(X))        # Same as sys.exc_info()[0], X.__class__
```

Because the `except's` `as` can be used to access the exception directly this way, `sys.exc_info` is more useful for *empty* `except` clauses that do not otherwise have a way to access the instance or its class. More importantly, well-designed programs usually should *not have to care* about which specific exception was raised at all—calling methods of the exception instance should automatically dispatch to behavior tailored for the exception raised.

There's more on this and `sys.exc_info` and its ilk in the next chapter. Also, see [Chapter 29](#) and [Part VI](#) at large if you've forgotten what `__class__` means in an instance, and the prior chapter for a review of the `as` used here.

Why Exception Hierarchies?

Because there are only three possible exceptions in the prior section's examples, it doesn't really do justice to the utility of class exceptions. In principle, we could achieve the same effects by coding a list of exception names in a parenthesized tuple within the `except` clause:

```
try:
    func()
except (General, Specific1, Specific2):    # Catch any of these
    ...
```

This approach may work for smaller, self-contained code. For large or high exception hierarchies, however, it will probably be easier to catch categories using class-based categories than to list every member of a category in a single `except` clause. Perhaps more importantly, you can extend exception hierarchies as software needs evolve by adding new subclasses without breaking existing handler code.

Suppose, for example, you code a numeric programming library in Python to be used by a large number of people. While you are writing your library, you identify two things that can go wrong with numbers in your code—division by zero and numeric overflow. You document these as the two standalone exceptions that your library may raise:

```
# mathlib.py
class Divzero(Exception): pass
class Oflow(Exception): pass

def func():
    ...
    raise Divzero()
...and so on...
```

Now, when people use your library, they typically wrap calls to your functions or classes in `try` statements that catch your two exceptions; after all, if they do not catch your exceptions, exceptions from your library will kill their code:

```
# client.py
import mathlib

try:
    mathlib.func()
except (mathlib.Divzero, mathlib.Oflow):
    ...handle and recover...
```

This works fine, and lots of people start using your library. Six months down the road, though, you revise it (as programmers are prone to do). Along the way, you identify a new thing that can go wrong—underflow, perhaps—and add that as a new exception:

```
# mathlib.py
class Divzero(Exception): pass
class Oflow(Exception): pass
class Uflow(Exception): pass
```

Unfortunately, when you re-release your code, you create a maintenance problem for your users. If they’ve listed your exceptions explicitly, they now have to go back and change every place they call your library to include the newly added exception name:

```
# client.py
try:
    mathlib.func()
except (mathlib.Divzero, mathlib.Oflow, mathlib.Uflow):
    ...handle and recover...
```

This may not be the end of the world. If your library is used only in-house, you can make the changes yourself. You might also ship a Python script that tries to fix such code automatically (it would probably be only a few dozen lines, and it would guess right at least some of the time). If many people have to change all their try statements each time you alter your exception set, though, this is not exactly the politest of upgrade policies.

Your users might try to avoid this pitfall by coding empty except clauses to catch *all* possible exceptions:

```
# client.py
try:
    mathlib.func()
except:                                     # Catch everything here (or catch Exception)
    ...handle and recover...
```

But as noted in the prior chapter, this workaround might catch more than they bargained for—things like running out of memory, keyboard interrupts (Ctrl+C), system exits, and even typos in their own try block’s code will all trigger exceptions, and such things should pass, not be caught and erroneously classified as library errors. Catching the `Exception` superclass improves on this but still intercepts—and thus may mask—program errors.

And really, in this scenario, users want to catch and recover from *only* the specific exceptions the library is defined and documented to raise. If any other exception occurs during a library call, it’s likely a genuine bug in the library (and it’s probably time to contact the vendor). As a rule of thumb, it’s usually better to be specific than general in exception handlers—an idea we’ll revisit as a “gotcha” in the next chapter.

So what to do, then? In principle again, the library module could provide a tuple object that contains all the exceptions it can possibly raise. The client could then import the tuple and name it in an except clause to catch all the library’s exceptions (recall that using a tuple catches any of its exceptions). This would work and support mods, but you’d need to keep the tuple up-to-date with library exceptions, and that’s both error-prone and tedious.

Class exception hierarchies solve this dilemma better. Rather than defining your library’s exceptions as a set of autonomous classes, arrange them into a class tree with a common superclass to encompass the entire category:

```
# mathlib.py
class NumErr(Exception): pass
class Divzero(NumErr): pass
class Oflow(NumErr): pass

def func():
    ...
    raise DivZero()
...and so on...
```

This way, users of your library simply need to list the common superclass (i.e., *category*) to catch all of your library’s exceptions—both now and in the future:

```
# client.py
import mathlib

try:
    mathlib.func()
except mathlib.NumErr:
    ...handle and recover...
```

When you go back and hack (update) your code again now, you can add new exceptions as new *sub-classes* of the common superclass:

```
# mathlib.py
...
class Uflow(NumErr): pass
```

The end result is that user code that catches your library’s exceptions will keep working, *unchanged*. In fact, you are free to add, delete, and change exceptions arbitrarily in the future—as long as clients name the superclass, and that superclass remains intact, they are insulated from changes in your exceptions set. In other words, class exceptions provide a better answer to maintenance issues than other solutions can.

Class-based exception hierarchies also support state retention and inheritance in ways that make them ideal in larger programs. To understand these roles, though, we first need to see how user-defined exception classes relate to the built-in exceptions from which they inherit.

Built-in Exception Classes

The prior section’s example wasn’t really pulled out of thin air. All built-in exceptions that Python itself may raise are predefined class objects. Moreover, they are organized into a shallow hierarchy with general superclass categories and specific subclass types, much like the prior section’s final exceptions class tree.

All the familiar exceptions you’ve seen (e.g., `SyntaxError`) are really just predefined classes, available as built-in names in the module named `builtins`. In addition, Python organizes the built-in exceptions into a hierarchy to support a variety of catching modes. For example:

BaseException: *topmost root, with printing and constructor defaults*

The top-level root superclass of exceptions. This class is not supposed to be directly inherited by user-defined classes (use `Exception` instead). It provides default printing and state retention behavior inherited by subclasses. If the `str` built-in is called on an instance of this class (e.g., by `print`), the class returns the display strings of the constructor arguments passed when the instance was created (or an empty string if there were no arguments). In addition, unless subclasses replace

this class's constructor, all of the arguments passed to this class at instance construction time are stored in its `args` attribute as a tuple.

Exception: root of user-defined exceptions

The top-level root superclass of application-related exceptions. This is an immediate subclass of `BaseException` and is a superclass to every other built-in exception, except the system exit event classes (`SystemExit`, `KeyboardInterrupt`, and `GeneratorExit`) and an exception-group class we'll ignore here. Nearly all user-defined classes should inherit from this class, not `BaseException`. When this convention is followed, naming `Exception` in a `try` statement's handler ensures that your program will catch everything but system exit events, which should normally be allowed to pass. In effect, `Exception` becomes a catchall in `try` statements but is more accurate than an empty `except`.

ArithmeticError: root of numeric errors

A subclass of `Exception`, and the superclass of all numeric errors. Its subclasses identify specific numeric errors: `OverflowError`, `ZeroDivisionError`, and `FloatingPointError`.

LookupError: root of indexing errors

A subclass of `Exception`, and the superclass category for indexing errors for both sequences and mappings: `IndexError` and `KeyError`.

OSError: root of IO and other system-function errors, with details

A subclass of `Exception`, with attributes for error details (e.g., `errno`, `strerror`, and `filename`), and subclasses for specific errors: `FileNotFoundError`, `PermissionError`, `TimeoutError`, and more.

And so on—because the built-in exception set is prone to frequent changes, this book doesn't document it exhaustively. You can read further about its contents and structure in the Python library manual.

Built-in Exception Categories

The built-in class tree allows you to choose how specific or general your handlers will be. For example, because the built-in exception `ArithmeticError` is a superclass for more specific exceptions such as `OverflowError` and `ZeroDivisionError`:

- By listing `ArithmeticError` in a `try`, you will catch *any* kind of numeric error raised.
- By listing `ZeroDivisionError`, you will intercept *just* that specific type of error and no others.

Similarly, because `Exception` is the superclass of all application-level exceptions, you can generally use it as a *catchall*—as outlined in the prior chapter, the effect is much like an empty `except`, but it allows system exit exceptions to pass and propagate as they usually should:

```
try:
    ...
except Exception:
    ...handle all application exceptions...           # Exits not caught here
else:
    ...handle no-exception case...
```

This technique is reliable because Python requires all classes to derive from built-in exceptions. Still, this scheme suffers most of the same potential pitfalls as the empty `except`, as described in the prior chapter—it might intercept exceptions intended for elsewhere, and it might mask genuine

programming errors. Since this is such a common issue, we'll revisit it one more time as a “gotcha” in the next chapter.

Whether or not you will leverage the categories in the built-in class tree, it serves as a good example. By using similar techniques for class exceptions in your own code, you can provide exception sets that are flexible and easily modified.

Default Printing and State

Built-in exceptions also provide default print displays and state retention, which is often as much logic as user-defined classes require. Unless you redefine the constructors your classes inherit from built-ins, any constructor arguments you pass to these classes are automatically saved in the instance's `args` tuple attribute and are automatically displayed when the instance is printed. An empty tuple and display string are used if no constructor arguments are passed, and a single argument displays as itself (not as a tuple) and serves as message details.

This explains why arguments passed to *built-in* exception classes show up in error messages—any constructor arguments are attached to the instance and displayed when the instance is printed:

```
>>> raise IndexError                                # Same as IndexError(): no arguments
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError

>>> raise IndexError('bad')                        # Constructor argument attached, printed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: bad

>>> i = IndexError('bad', 'stuff')                 # Available in object attribute "args"
>>> i.args
('bad', 'stuff')
>>> print(i)                                       # Displays args when printed manually
('bad', 'stuff')
>>> i                                             # Uses repr for echo, str for print
IndexError('bad', 'stuff')
```

The same holds true for *user-defined* exceptions because they inherit the constructor and display methods present in their built-in superclasses:

```
>>> class E(Exception): pass

>>> raise E
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
E

>>> raise E('bad')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
E: bad

>>> i = E('bad', 'stuff')
>>> i.args
('bad', 'stuff')
>>> print(i)
('bad', 'stuff')
>>> i
E('bad', 'stuff')
```


When intercepted in a try statement, the exception instance object gives access to both the original constructor arguments and the display method:

```
>>> try:
...     raise E('bad')                # Displays + saves constructor args
... except E as X:
...     print(f'{X} - {X.args} - {X!r}')
...
bad - ('bad',) - E('bad')

>>> try:
...     raise E('bad', 'stuff')       # Multiple args save/display a tuple
... except E as X:
...     print(f'{X} - {X.args} - {X!r}')
...
('bad', 'stuff') - ('bad', 'stuff') - E('bad', 'stuff')
```

Note that exception instance objects are not strings themselves, but use the `__str__` and `__repr__` operator-overloading methods we studied in [Chapter 30](#) to provide display strings for `print` and other contexts. To concatenate with real strings, perform manual conversions: `str(X)`, `'%s' % X`, `f'{X}'`, and the like.

Although this automatic state and display support is useful by itself, for more specific display and state retention needs, you can always redefine inherited methods such as `__str__` and `__init__` in Exception subclasses—as the next section shows.

Custom Print Displays

As we saw in the preceding section, by default, instances of class-based exceptions display whatever you passed to the class constructor when they are caught and printed:

```
>>> class MyBad(Exception): pass

>>> try:
...     raise MyBad('Sorry--my mistake!')
... except MyBad as X:
...     print(X)
...
Sorry--my mistake!
```

This inherited default display model is also used if the exception is displayed as part of an error message when the exception is not caught:

```
>>> raise MyBad('Sorry--my mistake!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MyBad: Sorry--my mistake!
```

For many roles, this is sufficient. To provide a more custom display, though, you can define one of two string-representation overloading methods in your class (`__repr__` or `__str__`) to return the string you want to display for your exception. The string the method returns will be displayed if the exception either is caught and printed or reaches the default handler:

```
>>> class MyBad(Exception):
...     def __str__(self):
...         return 'Stuff happens...'

>>> try:
...     raise MyBad()
... except MyBad as X:
...     print(X)
...
Stuff happens...

>>> raise MyBad()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MyBad: Stuff happens...
```

Whatever your method returns is included in error messages for uncaught exceptions and used when exceptions are printed explicitly. The method returns a hardcoded string here to illustrate, but it can also perform arbitrary text processing, possibly using state information attached to the instance object. The next section looks at state information options.

First, though, one fine point: you generally must redefine `__str__` for exception display purposes because the built-in exception superclasses already have a `__str__` method, and `__str__` is preferred to `__repr__` in some contexts—including error-message displays. If you define a `__repr__`, printing will happily call the built-in superclass's `__str__` instead:

```
>>> class Oops(Exception):
...     def __repr__(self): return 'Custom display not used'

>>> raise Oops("Nobody's perfect")
...
Oops: Nobody's perfect
```

But a custom `__str__` is used if defined:

```
>>> class Oops(Exception):
...     def __str__(self): return 'Custom display used'

>>> raise Oops("Nobody's perfect")
...
Oops: Custom display used
```

See [Chapter 30](#) for more details on these special operator-overloading methods.

Custom State and Behavior

Besides supporting flexible hierarchies, exception classes also provide storage for extra state information as instance *attributes*. As discussed earlier, built-in exception superclasses provide a default constructor that automatically saves constructor arguments in an instance tuple attribute named `args`. Although the default constructor is adequate for many cases, for more custom needs we can provide a constructor of our own. In addition, classes may define methods for use in handlers that provide precoded exception processing logic.

Providing Exception Details

When an exception is raised, it may cross arbitrary file boundaries—the `raise` statement that triggers an exception and the `try` statement that catches it may be in completely different module files. It is not generally feasible to store extra details in global variables because the `try` statement might not know which file the globals reside in. Passing extra state information along in the exception itself allows the `try` statement to access it more reliably.

With classes, this is nearly automatic. As we’ve seen, when an exception is raised, Python passes the class instance object along with the exception. Code in `try` statements can access the raised instance by listing an extra variable after the `as` keyword in an `except` handler. This provides a natural hook for supplying data and behavior to the handler. Generic instance-access tools like `sys.exc_info` used earlier enable the same interfaces.

For example, a program that parses data files might signal a formatting error by raising an exception instance that is filled out with extra details about the error (the input file here isn’t real because this demo dies before reading it!):

```
>>> class FormatError(Exception):
...     def __init__(self, line, file):          # Custom constructor
...         self.line = line
...         self.file = file
...
>>> def parser(file):                          # Parse the file first
...     raise FormatError(62, file=file)        # When an error is found
...
>>> try:
...     parser('code.py')
... except FormatError as X:
...     print(f'Error at: {X.file} #{X.line}')  # Custom state info
...
Error at: code.py #62
```

In the `except` clause here, the variable `X` is assigned a reference to the instance that was generated when the exception was raised. This gives access to the attributes attached to the instance by the custom constructor. Although we could rely on the default state retention of built-in superclasses, it’s less relevant to our application (and doesn’t support the keyword arguments used in the prior example):

```
>>> class FormatError(Exception): pass          # Inherited constructor
...
>>> def parser(file):
...     raise FormatError(file, 62)             # No keywords allowed!
...
>>> try:
...     parser('code.py')
... except FormatError as X:
...     print(f'Error at: {X.args[0]} #{X.args[1]}') # Generic state info
...
Error at: code.py #62
```

Providing Exception Methods

Besides enabling application-specific state and display, classes also support extra behavior for exception objects. That is, the exception class can also define unique *methods* to be called in handlers. The file `parsely.py` in [Example 35-3](#), for example, adds a method that uses exception custom state information to log errors to a file automatically.

Example 35-3. *parsely.py*

```
from time import asctime

class FormatError(Exception):
    logfile = 'parser-errors.txt'
    def __init__(self, line, file):
        self.line = line
        self.file = file
    def logerror(self):
        with open(self.logfile, 'a') as log:
            print(f'Error at: {self.file} #{self.line} [{asctime()}]', file=log)

def parser(file):
    # Parse a file here...
    raise FormatError(line=62, file=file)

if __name__ == '__main__':
    try:
        parser('code.py')
    except FormatError as exc:
        exc.logerror()
```

When run, this script appends its error message to a file in response to method calls in the exception handler (use `type` instead of the Unix `cat` on Windows, and see Python manuals for `time.asctime`):

```
$ python3 parsely.py
$ python3 parsely.py
$ cat parser-errors.txt
Error at: code.py #62 [Sat Jul 13 12:22:19 2024]
Error at: code.py #62 [Sat Jul 13 12:22:25 2024]
```

In such a class, methods (like `logerror`) may also be inherited from superclasses, and instance attributes (like `line` and `file`) provide a place to save state information that provides extra context for use in later method calls. Moreover, exception classes are free to customize and extend inherited behavior:

```
class CustomFormatError(FormatError):
    def logerror(self):
        ...something unique here...

...
raise CustomFormatError(...)

...
try:
    ...
except FormatError as exc:
    exc.logError()           # Runs the raised class's version
```

In other words, because they are defined with classes, all the benefits of OOP that we studied in [Part VI](#) are available for use with exceptions in Python.

Two final notes here: first, the raised instance object assigned to `exc` in this code is also available generically as the second item in the result tuple of the `sys.exc_info()` call used earlier—a tool that returns information about the exception being handled. This call can be used if you do not list an exception name in an `except` clause but still need access to the exception that occurred, or to any of its attached state information or methods. And second, although our class's `logerror` method appends a custom message to a logfile, it could also generate Python's standard error message with stack trace using tools in the `traceback` standard-library module, which uses `traceback` objects.

To learn more about `sys.exc_info` and `tracebacks`, though, we need to move ahead to the next chapter.

Exception Groups: Yet Another Star!

But wait—just when you thought it was safe to put exceptions in the `win` column, the exceptions story has recently sprouted yet another wild plot twist, which is sufficiently limited and arcane to pass as an optional follow-up topic for most Python learners (and was deferred until now for this reason). Lest it crop up in one of those silly job interviews that favor the inane over the practical, though, here's a quick peek.

Let's get right to the code. As we've seen, `try` statements normally run at most *one* matching handler clause, plus an optional `finally` on exit:

```
>>> try:
...     raise IndexError()
... except IndexError:
...     print('Got IE')
... except (SyntaxError, TypeError):
...     print('Got SE')
...
Got IE
```

Python 3.11, though, adds the ability to trigger *multiple* matching handlers in a single `try` statement by wrapping them in an *exception group* and catching them with `except*` clauses:

```
>>> try:
...     raise ExceptionGroup('Many', [IndexError(), SyntaxError()])
... except* IndexError:
...     print('Got IE')
... except* (SyntaxError, TypeError):
...     print('Got SE')
...
Got IE
Got SE
```

In a nutshell, each `except*` clause can process and consume one exception, or one batch of them, in the group. Here, the first clause runs for `IndexError` and the second for `SyntaxError` (a tuple means “any” as before). The *group* is simply an exception object made by calling a built-in exception class provided for this role, passing a message-string label used in displays, along with a sequence of exceptions to be matched by `except*` clauses in a `try`.

Syntactically, an empty `except*` is not allowed, and a basic `except` cannot be mixed with `except*`—but an `else` and `finally` can. Moreover, `except*` cannot host a `break`, `continue`, or `return`—but `except` can. Like the awkward `except/else/finally` mixing rules before it, these special cases probably qualify `except*` as a distinct statement form; adding it to the mix makes `try` an overloaded jumble of semi-related functionality.

Semantically, each `except*` clause executes at most once, and consumes all matching exceptions in the group. In addition, each exception in the group is handled by at most one `except*` clause—the topmost clause that matches it; optional `as` variables are assigned exception groups—with attributes like `exceptions` that expose their contents; and any *unmatched* exceptions in the group are reraised after the `try` statement processes matches—with a top-level message that denotes those unmatched:

```

>>> excs = ExceptionGroup('Many', [IndexError(), SyntaxError(), TypeError()])
>>> try:
...     raise excs
... except* IndexError as E:
...     print(f'Got IE: {E} => {E.exceptions}')
... except* SyntaxError as E:
...     print(f'Got SE: {E} => {E.exceptions}')
...
Got IE: Many (1 sub-exception) => (IndexError(),)
Got SE: Many (1 sub-exception) => (SyntaxError(),)
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 2, in <module>
| ExceptionGroup: Many (1 sub-exception)
+-+----- 1 -----
| TypeError
+-----

```

When the group has multiple exceptions of the *same* type, a matching `except*` consumes them all and can process them in normal iteration code. As also shown next, *spaces* around the `*` are allowed and ignored—despite all the `except*` labels in docs, and more reflective of the wildcard nature of these clauses:

```

>>> try:
...     raise ExceptionGroup('Dups', [IndexError(), TypeError(1), TypeError(2)])
... except *IndexError:
...     print('Got IE')
... except *TypeError as E:
...     print(f'Got TE: {E} => {E.exceptions}')
...
Got IE
Got TE: Dups (2 sub-exceptions) => (TypeError(1), TypeError(2))

```

As usual, a group's exception matches an `except*` that names its class or one of its *superclasses*. Because of this, the *ordering* of clauses is both subtle and important—the first match wins and removes exceptions from the group. In the following, for example, the first clause gobbles `IndexError` via its `LookupError` superclass, along with the two `TypeError`s in the group (but reversing the clauses' order would handle `IndexError` separately):

```

>>> try:
...     raise ExceptionGroup('Dups', [IndexError(), TypeError(1), TypeError(2)])
... except* (TypeError, LookupError) as E:
...     print('Got1:', E)
... except* IndexError as E:
...     print('Got2:', E)
...
Got1: Dups (3 sub-exceptions)

```

The `except*` can also match basic *individual* exceptions, which are automatically wrapped in a group to appease group-based code in the handler:

```

>>> try:
...     raise IndexError
... except* IndexError as E:
...     print(f'Got IE: {E} => {E.exceptions}')
...
Got IE: (1 sub-exception) => (IndexError(),)

```

And a basic `except` can catch a group as a *collective* and process it manually, but an `except*` cannot catch a group because it would be ambiguous (a schism of the sort that’s usually a hallmark of an ad hoc extension):

```
>>> try:
...     raise ExceptionGroup('Lots', [IndexError(), SyntaxError()])
... except Exception as E:
...     print(f'Got group: {E} => {E.exceptions}')
...     for exc in E.exceptions:
...         print('With exc:', type(exc))
...
Got group: Lots (2 sub-exceptions) => (IndexError(), SyntaxError())
With exc: <class 'IndexError'>
With exc: <class 'SyntaxError'>
```

Finally, for comparison, here’s a *catchall* in both models—though there’s no reason to use `except*` to catch a single exception (unless overly complicated code is your thing):

```
>>> try:
...     raise ExceptionGroup('Dups', [IndexError()])
... except* Exception as E:
...     print(type(E.exceptions[0]))
...
<class 'IndexError'>

>>> try:
...     raise IndexError()
... except Exception as E:
...     print(type(E))
...
<class 'IndexError'>
```

For another exception-groups example, see the next chapter’s [Example 36-2](#), which demos how runtime nesting consumes items in groups (short story: groups propagate until empty, then die like individual exceptions).

Design concerns aside, the “why” of `except*` is even more elusive than the “how.” While it’s conceivable that some programs may wish to collect a set of exceptions and send them to a `try` statement as a *batch*, it’s harder to understand why these wildly rare programs could not be expected to package with normal exception objects and process with normal iteration code—instead of convoluting the Python language for everyone.

Because exception groups are an obscure tool with very narrow roles, we’ll defer to Python’s manuals for more info on this esoteric `try` extension that, like many a Python mod, seems to blow up complexity radically to address a purported need that somehow managed to go unnoticed for all of Python’s first three decades+. How did we live?

Chapter Summary

In this chapter, we explored both built-in exceptions and ways to code exceptions of our own. As we learned, exceptions are implemented as class instance objects. Exception classes support the concept of exception hierarchies that ease maintenance, allow data and behavior to be attached to exceptions as instance attributes and methods, and allow exceptions to inherit tools from superclasses as usual in OOP.

We saw that in a `try` statement, catching a superclass catches that class as well as all subclasses below it in the class tree—superclasses become exception category names, and subclasses become more specific

exception types within those categories. We also saw that the built-in exception superclasses we must inherit from provide usable defaults for printing and state retention, which we can override if desired.

Finally, armed with our new knowledge of exception objects, we also peeked at exception groups and the `except*` clause, used to run multiple handlers in a `try`. We questioned whether this extension's convolution of `try` statements is justified by its perceived roles; it's a lot to ask of most Python users, but this question is ultimately yours to answer.

The next chapter wraps up this part of the book by exploring some common use cases for exceptions and surveying tools commonly used by Python programmers. Before we get there, though, here's this chapter's quiz.

Test Your Knowledge: Quiz

1. What are the two main constraints on user-defined exceptions in Python?
2. How are raised exceptions matched to `except` handler clauses?
3. Name two ways that you can attach context information to exception objects.
4. Name two ways that you can specify the error-message text for exception objects.
5. What do `except*` clauses do in a `try` statement?

Test Your Knowledge: Answers

1. Exceptions must be defined by *classes* (that is, a class instance object is raised and caught). In addition, exception classes must be derived from the *built-in* class `BaseException` or one of its subclasses; most programs inherit from its `Exception` subclass to support catchall handlers for normal kinds of exceptions.
2. Exceptions match by superclass relationships: naming a *superclass* in an exception handler will catch instances of that class, as well as instances of any of its *subclasses* lower in the class tree. Because of this, you can think of superclasses as general exception categories and subclasses as more specific types of exceptions within those categories.
3. You can attach context information to exceptions with either custom or default constructors. A *custom* constructor can fill out attributes in a raised instance object that are specific to the program. For simpler needs, built-in exception superclasses provide a *default* constructor that stores its arguments on the instance automatically as tuple attribute `args`. Handlers can list a variable to be assigned to the raised instance, then go through this name to access attached state information and call any methods defined in the class.
4. The error-message text in exceptions can be specified with a custom `__str__` operator-overloading method. For simpler needs, built-in exception superclasses automatically display anything you pass to the class constructor. Operations like `print` and `str` automatically fetch the display string of an exception object when it is printed either explicitly or as part of an error message.
5. In a `try`, `except*` is used to run possibly *multiple* handlers for exceptions raised as part of a *group*. The `except*` also comes with heavy semantics, has special-case syntax and rules, does not combine with basic `except`, and is rarely useful in most Python programs. Nevertheless, there it is.

Exception Odds and Ends

This chapter rounds out this part of the book with the usual collection of stray topics, common-usage examples, and design concepts, followed by this part’s gotchas and exercises. Because this chapter also closes out the fundamentals portion of the book at large, it includes a brief overview of concepts and development tools to help as you make the transition from Python language beginner to Python application developer.

Nesting Exception Handlers

Most of our examples so far have used only a single `try` to catch exceptions, but what happens if one `try` is physically nested inside another? For that matter, what does it mean if a `try` calls a function that runs another `try`? Technically, `try` statements can nest in terms of both syntax and the runtime control flow through your code. This was mentioned earlier in brief but merits clarification here.

Both of these cases can be understood if you realize that Python *stacks* `try` statements at runtime. When an exception is raised, Python returns to the most recently entered `try` statement with a matching `except` clause. Because each `try` statement leaves a marker on the top of a LIFO stack, Python can jump back to earlier `trys` by inspecting the stacked markers. This nesting of active handlers is what we mean when we talk about propagating exceptions up to “higher” handlers—such handlers are simply `try` statements entered *earlier* in the program’s execution flow.

Figure 36-1 illustrates what occurs when `try` statements with `except` clauses nest at runtime. The amount of code that goes into a `try` block can be substantial, and it may contain function calls that invoke other code watching for the same exceptions. When an exception is eventually raised, Python jumps back to the most recently entered `try` statement that names that exception, runs that statement’s `except` clause, and then resumes execution after that `try`.

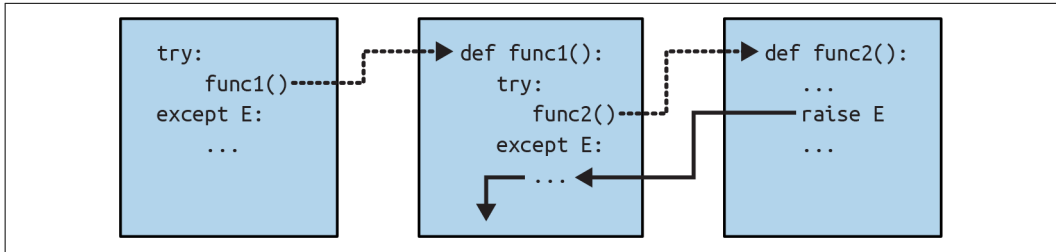


Figure 36-1. Nested *try/except* combinations

Once the exception is caught, its life is over—control does not jump back to *all* matching *try*s that name the exception; only the first (i.e., most recent) one is given the opportunity to handle it. In [Figure 36-1](#), for instance, the `raise` statement in the function `func2` sends control back to the handler in `func1`, and then the program continues within `func1`.

By contrast, when *try* statements that contain only *finally* clauses are nested, *each* *finally* block is run in turn when an exception occurs—Python continues propagating the exception up to other *try*s, and eventually perhaps to the top-level default handler (the standard error-message printer). As [Figure 36-2](#) illustrates, the *finally* clauses do not kill the exception—they just specify code to be run on the way out of each *try* during the exception propagation process. If there are many *try/finally* combos active when an exception occurs, they will *all* be run unless an *except* clause catches the exception somewhere along the way.

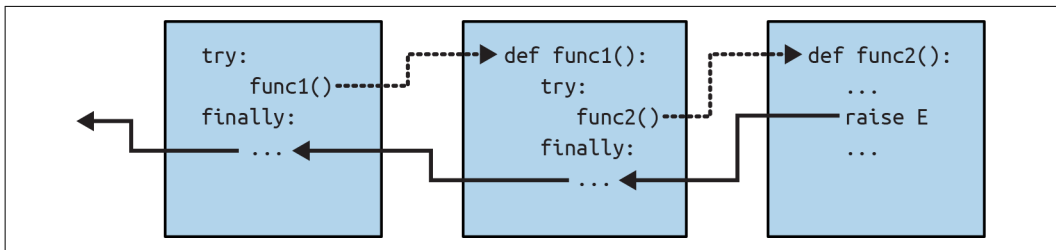


Figure 36-2. Nested *try/finally* combinations

In other words, where the program goes when an exception is raised depends entirely upon *where it has been*—it's a function of the runtime flow of control through the script, not just its syntax. The propagation of an exception essentially proceeds backward through time to *try* statements that have been entered but not yet exited. This propagation stops as soon as control is unwound to a matching *except* clause, but not as it passes through *finally* clauses on the way.

The prior chapter's `except*` clauses don't change this story—if they consume every exception in the raised group, the aggregate exception ends in the *try* as usual. As we saw, unmatched exceptions are reraised after the `except*` clauses have their chance and are propagated on to other *try* statements or the top-level handler, but this is not fundamentally different from exceptions unmatched by an *except*.

Example: Control-Flow Nesting

Let's turn to examples to make this nesting concept more concrete. The module file in [Example 36-1](#) defines three functions: `action1` wraps a call to `action2` in a *try* handler, `action2` does likewise for a

call to `action3`, and `action3` is coded to trigger a built-in `TypeError` exception (you can't add numbers and sequences).

Example 36-1. `nested_exc_normal.py`

```
def action3():
    print(1 + [])          # Generate TypeError

def action2():
    try:                   # Most recent matching try
        action3()
    except TypeError:
        print('Inner try') # Match kills the exception
        raise              # Unless manually reraised

def action1():
    try:
        action2()
    except TypeError:
        print('Outer try') # Run only if action2 reraises

if __name__ == '__main__': action1()
```

Notice, though, that when `action3` triggers the exception, there will be *two* active `try` statements—the older one in `action1` and the newer one in `action2`. Python picks and runs just the most recent `try` with a matching `except`—which in this case is the `try` inside `action2`. In this demo, `action2` also manually reraises the `TypeError` with `raise` to trigger the `try` in `action1`, but the exception would otherwise die in `action2`:

```
$ python3 nested_exc_normal.py
Inner try
Outer try
```

The same happens for an exception *group*, though the exception doesn't die until the entire group has been matched. In [Example 36-2](#), for instance, `action2` picks off `IndexError`, `action1` consumes `TypeError`, and `SyntaxError` propagates to the top-level default handler (or an earlier matching `try`, if one had been run).

Example 36-2. `nested_exc_group.py`

```
def action3():
    raise ExceptionGroup('Nest*', [IndexError(1), TypeError(2), SyntaxError(3)])

def action2():
    try:
        action3()
    except* IndexError:      # Consume matches, rest propagate
        print('Got IE')

def action1():
    try:
        action2()
    except* TypeError:      # Consume matches, rest propagate
        print('Got TE')

if __name__ == '__main__': action1()
```

When run, each function's try consumes exceptions it matches, and the top-level handler prints an error message for the last remaining unmatched item in the group:

```
$ python3 nested_exc_group.py
Got IE
Got TE
+ Exception Group Traceback (most recent call last):
...etc...
| ExceptionGroup: Nest* (1 sub-exception)
+----- 1 -----
| SyntaxError: 3
+-----
```

Whether groups or individual exceptions are raised, the place where an exception winds up jumping to depends on the control flow through the program at runtime. Because of this, to know where you will go, you need to know where you've been. In other words, routing for exceptions nested at runtime is more a function of control flow than of statement syntax. That said, we can also nest exception handlers syntactically—an equivalent case we turn to next.

Example: Syntactic Nesting

As discussed when we studied clause combinations of the try statement in [Chapter 34](#), it is also possible to nest try statements syntactically by their position in your source code:

```
>>> from nested_exc_normal import action3

>>> try:
...     try:
...         action3()
...     except TypeError:      # Most-recent matching try
...         print('Inner try')
...         raise
... except TypeError:         # Here, only if nested handler reraises
...     print('Outer try')
...
Inner try
Outer try
```

Really, though, this code just sets up the same handler-nesting structure as, and behaves identically to, the try statements in [Example 36-1](#). In fact, syntactic nesting works just like the cases sketched in [Figures 36-1](#) and [36-2](#). The only difference is that the nested handlers are physically embedded in a try block, not coded elsewhere in functions that are called from the try block. For example, nested finally handlers all fire on an exception, whether they are nested syntactically or by means of the runtime flow through physically separated parts of your code:

```
>>> try:
...     try:
...         action3()
...     finally:
...         print('Inner try')
... finally:
...     print('Outer try')
...
Inner try
Outer try
Traceback (most recent call last):
...etc...
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

See [Figure 36-2](#) for a graphic illustration of this code’s operation; the effect is the same, but the function logic has been *inlined* as nested statements here. As a more comprehensive example of syntactic nesting at work, consider the file listed in [Example 36-3](#).

Example 36-3. except-finally.py

```
def raise1(): raise IndexError
def noraise(): return
def raise2(): raise SyntaxError

for func in (raise1, noraise, raise2):
    print(f'<{func.__name__}>')
    try:
        try:
            func()
        except IndexError:
            print('caught IndexError')
    finally:
        print('finally run')
    print('...')
```

This code catches an exception if a matching one is raised and performs a `finally` termination-time action regardless of whether an exception occurs. This may take a few moments to digest, but the effect is the same as combining an `except` and a `finally` clause in a single `try` statement:

```
$ python3 except-finally.py
<raise1>
caught IndexError
finally run
...
<noraise>
finally run
...
<raise2>
finally run
Traceback (most recent call last):
...etc...
SyntaxError: None
```

As we saw in [Chapter 34](#), `except` and `finally` clauses can be mixed in the same `try` statement. While this, along with multiple `except` clauses, makes the syntactic nesting shown in this section largely academic, the equivalent runtime nesting is common in larger Python programs. Moreover, syntactic nesting can make the disjoint roles of `except` and `finally` explicit and might be useful for implementing alternative exception-handling behaviors.

Exception Idioms

We’ve seen the mechanics behind exceptions. Now, let’s survey the ways they are typically used. Some of these are reviews of roles we’ve explored in earlier chapters, collected here as part of a referable set.

Breaking Out of Multiple Nested Loops: “go to”

As mentioned at the start of this part of the book, exceptions can often be used to serve the same roles as other languages’ “go-to” statements to implement more arbitrary control transfers. Exceptions, however, provide a more structured option that localizes the jump to a specific block of nested code.

In this role, `raise` is like “go to,” and `except` clauses and exception names take the place of program labels. You can only jump out of code wrapped in a `try` this way, but that’s a crucial feature—truly arbitrary “go to” statements can make code extraordinarily difficult to understand and maintain (“spaghetti code” in developer lingo).

For example, Python’s `break` statement exits just the single closest enclosing loop, but we can always use exceptions to break out of more than one loop level if needed, as in [Example 36-4](#).

Example 36-4. `breaker.py`

```
class Exitloop(Exception): pass

try:
    while True:
        while True:
            for i in range(10):
                if i > 3: raise Exitloop          # break exits just one level
                print('loop3: %s' % i)          # raise can exit many
            print('loop2')
        print('loop1')
except Exitloop:
    print('continuing')                        # Or just pass, to move on

print(f'{i=}')                               # Loop variable not undone
```

When run, the `raise` in the `for` breaks out of three nested loops immediately:

```
$ python3 breaker.py
loop3: 0
loop3: 1
loop3: 2
loop3: 3
continuing
i=4
```

If you change the `raise` in this to `break`, you’ll get an infinite loop because you’ll break only out of the most deeply nested `for` loop, and wind up in the second-level `while` loop nesting. The code would then print “loop2” and start the `for` again. Make the mod to see for yourself—but get ready to type Ctrl+C to stop the code!

Also, notice that variable `i` is still what it was in `for` after the `try` statement exits. As previously noted, variable assignments made in a `try` are not undone in general, though as we’ve seen, exception instance variables listed in `except` clause as headers are localized to that clause, and the local variables of any functions that are exited as a result of a `raise` are discarded. Technically, active functions’ local variables are popped off the call stack, and the objects they reference may be garbage-collected as a result, but this is an automatic step.

Exceptions Aren’t Always Errors

In Python, all errors are exceptions, but not all exceptions are errors. For instance, we saw in [Chapter 9](#) that file object `read` methods return an empty string at the end of a file. In contrast, the built-in `input` function—which we first met in [Chapter 3](#) and deployed in an interactive loop in [Chapter 10](#)—reads a line of text from the standard input stream, `sys.stdin`, on each call and raises the built-in `EOFError` at end-of-file.

Unlike file methods, this function does not return an empty string—an empty string from `input` means an empty line. Despite its name, though, the `EOFError` exception is just a *signal* in this context, not an error. Because of this behavior, unless the end-of-file should terminate a script, `input` often appears wrapped in a try handler and nested in a loop, as in the following code:

```
while True:
    try:
        line = input()          # Read line from stdin
    except EOFError:
        break                  # Exit loop at end-of-file
    else:
        ...process next line here...
```

Several other built-in exceptions are similarly signals, not errors—for example, calling `sys.exit()` and pressing Ctrl+C on your keyboard raise `SystemExit` and `KeyboardInterrupt`, respectively.

Python also has a set of built-in exceptions that represent *warnings* rather than errors; some of these are used to signal the use of deprecated (soon to be phased out) language features. See the standard-library manual’s description of built-in exceptions for more information, and consult the `warnings` module’s documentation for more on exceptions raised as warnings.

Functions Can Signal Conditions with raise

User-defined exceptions can also signal nonerror conditions. For instance, a search routine can be coded to raise an exception when a match is found instead of returning a status flag for the caller to interpret. In the following abstract code, the try/except/else exception handler does the work of an if/else return-value tester:

```
class Found(Exception): pass

def searcher():
    if ...success...:
        raise Found()          # Raise exceptions instead of returning flags
    else:
        return

try:
    searcher()
except Found:
    ...success...               # Exception if item was found
else:
    ...failure...              # else returned: not found
```

More generally, such a coding structure may also be useful for any function that cannot return a *sentinel value* to designate success or failure. In a widely applicable function, for instance, if all objects are potentially valid return values, it’s impossible for any return value to signal a failure condition. Exceptions provide a way to signal results without a return value:

```
class Failure(Exception): pass

def searcher():
    if ...success...:
        return founditem
    else:
        raise Failure()

try:
    item = searcher()
```

```
except Failure:
    ...not found...
else:
    ...use item here...
```

Because Python is dynamically typed and polymorphic to the core, exceptions, rather than sentinel return values, are the generally preferred way to signal such conditions.

Closing Files and Server Connections

We encountered examples in this category in [Chapter 34](#). As a review, exception processing tools are also commonly used to ensure that system resources are finalized, regardless of whether an error occurs during processing or not.

For example, some servers require connections to be closed in order to terminate a session. Similarly, output files may require close calls to flush their buffers to disk for waiting consumers; input files may consume file descriptors if not closed; and CPython closes open files when garbage-collecting them, but this isn't always predictable or reliable.

As we saw in [Chapter 34](#), the most general and explicit way to guarantee termination actions for a specific block of code is the `try/finally` combination:

```
myfile = open('somefile', 'w')
try:
    ...process myfile...
finally:
    myfile.close()
```

As we also saw, some objects make this potentially easier by providing *context managers* that terminate or close resources for us automatically when run by the `with` statement:

```
with open('somefile', 'w') as myfile:
    ...process myfile...
```

If you want to know which option is better, flip back to “[The Termination-Handlers Shoot-Out](#)” on [page 908](#)—and draw your own conclusions.

Debugging with Outer `try` Statements

You can also make use of exception handlers to replace Python's default top-level exception-handling behavior. By wrapping an entire program (or a call to it) in an outer `try` in your top-level code, you can catch any exception that may occur while your program runs, thereby subverting the default program termination.

In the following, the empty `except` clause catches any uncaught exception raised while the program runs. To get hold of the actual exception that occurred in this mode, fetch the `exc_info` function call result from the built-in `sys` module; it returns a tuple whose first two items contain the currently handled exception's class and the instance object raised (more on `sys.exc_info` in a moment):

```
try:
    ...run program...
except:
    # All uncaught exceptions come here
    import sys
    print('uncaught!', sys.exc_info()[0], sys.exc_info()[1])
```


This structure is commonly used during development to keep programs active even after errors occur. It's also used when testing other program code, as described in the next section: coded within a loop, this structure allows you to run additional tests without having to restart.

Running In-Process Tests

Some of the coding patterns we've just seen can be combined in a test-driver script that tests other code imported and run within the same process (i.e., program run). The following partial and abstract code sketches the general model:

```
import sys
log = open('testlog', 'a')
from testapi import moreTests, runNextTest, testName
def testdriver():
    while moreTests():
        try:
            runNextTest()
        except:
            print('FAILED', testName(), sys.exc_info()[2], file=log)
        else:
            print('PASSED', testName(), file=log)
testdriver()
```

The `testdriver` function here cycles through a series of test calls. Because an uncaught exception in any of them would normally kill this test driver, tests are wrapped in a `try` to continue the testing process if a test fails. The empty `except` catches any uncaught exception generated by a test case and uses `sys.exc_info` to log the exception to a file. The `else` clause is run when no exception occurs—the test success case.

Such boilerplate code is typical of systems that test imported functions, modules, and classes. In practice, though, testing can be much more sophisticated. For instance, to test *external programs*, you could instead check status codes or outputs generated by program-launching tools such as `os.system` and `os.popen`, which were used earlier in this book and are covered in Python's standard-library manual. Such tools do not generally raise exceptions for errors in the external programs—in fact, the test cases may run in parallel with the test driver.

At the end of this chapter, we'll also briefly explore more complete testing frameworks provided by Python, such as `doctest` and `PyUnit`, which provide tools for comparing expected outputs with actual results.

More on `sys.exc_info`

The `sys.exc_info` result used in the last two sections allows an exception handler to generically gain access to the exception being handled. This is especially useful when using the empty `except` clause to catch everything blindly because it allows you to determine what was raised:

```
try:
    ...
except:
    # sys.exc_info()[0:2] are the exception class and instance
```

If no exception is being handled, this call returns a tuple containing three `None` values. Otherwise, the values returned are (*type*, *value*, *traceback*), where:

- *type* is the class of the exception being handled.
- *value* is the class instance that was raised.
- *traceback* is a traceback object that represents the call stack at the point where the exception originally occurred, and may be used by the `traceback` module to generate error messages.

As we saw in [Chapter 35](#), `sys.exc_info` can also sometimes be useful to determine the specific exception type when catching exception category superclasses. As we’ve also learned, though, because in this case you can also get the exception type by fetching the `__class__` attribute or type result of the instance obtained with the `as` clause, `sys.exc_info` is rarely useful outside the empty `except`:

```
try:
    ...
except General as instance:
    # instance.__class__ or type(instance) is the exception class
    # but instance.method() does the right thing for this instance
```

As we’ve seen, using `Exception` for the *General* exception name here would catch all nonexit exceptions; it’s similar to an empty `except` but less extreme and still gives access to the exception instance and its class. Even so, leveraging *polymorphism* by calling the instance’s *methods* is often a better approach than testing exception types.

The `sys.exception` alternative—and `diss`

As yet another option, a new call added in Python 3.11, `sys.exception`, returns *just* the exception instance raised—the same object assigned to the variable listed after `as` in `except` clauses, and equivalent to the second item in the `sys.exc_info` result (i.e., `sys.exc_info()[1]`). Hence, the following work the same in a `try`:

```
except ...:
    print('uncaught!', sys.exc_info()[0], sys.exc_info()[1])

except ...:
    print('uncaught!', type(sys.exception()), sys.exception())
```

More generally, there are now *three* ways to obtain the same information about a caught exception in `try` (two of which in the following are nested in a tuple to match `exc_info` and display with `repr` instead of `str`):

```
>>> class E(Exception): pass
...
>>> try:
...     raise E('info')
... except E as X:
...     print((type(X), X))
...     print(sys.exc_info()[2])
...     print((type(sys.exception()), sys.exception()))
...
(<class '__main__.E'>, E('info'))
(<class '__main__.E'>, E('info'))
(<class '__main__.E'>, E('info'))
```

When using `sys.exception`, the exception class is available from the instance via `__class__` or `type` (as shown), and the traceback is normally present in the exception instance's `__traceback__` object. Though largely trivial, the new call avoids an index or slice when only the instance is needed.

Less pleasantly, with the addition of `sys.exception`, the `sys.exc_info` call has also been branded “old-style” in Python’s docs, but doing this for the sake of a redundant call added just over a year ago seems both opinionated and divisive—if not software ageism. Naturally, you’re welcome and encouraged to use either call in your code, but this book generally recommends tools that are traditional, common, and inclusive.

Displaying Errors and Tracebacks

Finally, the exception traceback object available in the prior section’s `sys.exc_info` data is also used by the standard library’s `traceback` module to generate the standard error message and stack display manually. This module has a handful of interfaces that support wide customization, which we don’t have space to cover usefully here, but the basics are simple. Consider the (judgmentally named) file in [Example 36-5](#), *badly.py*.

Example 36-5. badly.py

```
import traceback

def inverse(x):
    return 1 / x

try:
    inverse(0)
except Exception:
    traceback.print_exc(file=open('badly.txt', 'w'))
print('Bye')
```

This code uses the `print_exc` convenience function in the `traceback` module, which internally uses `sys.exc_info` data (technically, it was changed to use the `sys.exception` component as part of the prior section’s subjective purge). When run, the script prints the standard error message to a file—useful in programs that need to catch errors but still record them in full (again, `type` is the Windows equivalent of Unix `cat` here):

```
$ python3 badly.py
Bye

$ cat badly.txt
Traceback (most recent call last):
  File ".../LP6E/Chapter36/badly.py", line 7, in <module>
    inverse(0)
  File ".../LP6E/Chapter36/badly.py", line 4, in inverse
    return 1 / x
    ~~~~
ZeroDivisionError: division by zero
```

For much more on traceback objects, the `traceback` module that uses them, and related topics, consult your favorite Python reference resources.

Exception Design Tips and Gotchas

This chapter is lumping design tips and gotchas together because it turns out that the most common exception gotchas stem from design issues. By and large, exceptions are easy to use in Python. The real art behind them is in deciding how specific or general your `except` clauses should be and how much code to wrap up in `try` statements. Let's address the latter of these choices first.

What Should Be Wrapped

In principle, you could wrap every statement in your script in its own `try`, but that would just be silly (the `try` statements would then need to be wrapped in `try` statements!). What to wrap is really a design issue that goes beyond the language itself, and it will become more apparent with use. But as a summary, here are a few rules of thumb:

- Operations that commonly fail should generally be wrapped in `try` statements. For example, operations that interface with system state (file opens, socket calls, and the like) are prime candidates for `try`.
- Unless they should fail—in a simple script, you may *want* failures to kill your program instead of being caught and ignored, especially if the failure is a showstopper. Failures in Python normally generate useful error messages instead of hard crashes, and this is the best outcome some programs could hope for.
- Cleanup actions that must be run regardless of exception outcomes should generally be run with a `try/finally` combination unless a context manager is available as a `with` option.
- Wrapping the *call* to a function in a single `try` statement often makes for less code than wrapping operations in the function itself. That way, all exceptions in the function percolate up to the single `try` around the call.

The types of programs you write will probably influence the amount of exception handling you code as well. Servers, test runners, and GUIs, for instance, must generally catch and recover from exceptions. Simpler one-shot scripts, though, will often ignore exception handling completely because failure at any step requires shutdown.

In all cases, keep in mind that failures in Python normally generate useful error messages instead of hard crashes. Even without `try`, this is often a better outcome than some programs could hope for.

Catching Too Much: Avoid Empty `except` and `Exception`

As we've learned, Python lets us pick and choose which exceptions to catch, but it's important not to be too inclusive. For example, we've seen that an empty `except` clause catches every exception. That's easy to code and sometimes desirable, but it may also wind up intercepting an error that's expected by a `try` elsewhere:

```
def func():
    try:
        ...
    except:
        ...

try:
    func()
```

IndexError is raised in here
But everything comes here and dies!

```
except IndexError:          # Exception should be processed here
    ...
```

Perhaps worse, such code might also catch unrelated critical exceptions. Even things like memory errors, program typos, iteration stops, keyboard interrupts, and system exits raise exceptions in Python. Unless you're writing a debugger or similar tool, such exceptions should not usually be intercepted in your code.

For example, scripts normally exit when control falls off the end of the top-level file, but Python also provides a built-in `sys.exit(statuscode)` call to allow early terminations. This works by raising a built-in `SystemExit` exception to end the program so that `try/finally` handlers run on the way out and tools can intercept the event. Because of this, a `try` with an empty `except` might unknowingly prevent an exit, as in [Example 36-6](#).

Example 36-6. *exiter.py*

```
import sys

def bye():
    sys.exit(62)          # Crucial error: abort now!

try:
    bye()
except:
    print('Got it')       # Oops--we ignored the exit

print('Continuing...')
```

When run, the script happily keeps going after a call to shut it down:

```
$ python3 exiter.py
Got it
Continuing...
```

You simply might not expect all the kinds of exceptions that could occur during an operation. Per the prior chapter, using the built-in `Exception` superclass can help because it is not a superclass of `SystemExit`:

```
try:
    bye()
except Exception:         # Won't catch exits, but _will_ catch many others
    ...
```

In some cases, though, this scheme is no better than an empty `except` clause—because `Exception` is a superclass above all built-in exceptions except system-exit events, it still has the potential to catch exceptions meant for elsewhere in the program. Worse, using *either* the empty `except` or `Exception` will also catch programming errors, which should usually be allowed to pass. In fact, these two techniques can effectively *turn off* Python's error-reporting machinery, making it difficult to notice mistakes in your code. Consider this code, for example:

```
mydictionary = {...}
...
try:
    x = myditctionary[key]    # Oops: misspelled name
except:
    x = None                 # Assume we got KeyError - only - here
...continue here with x...
```

The coder here assumes that the only sort of error that can happen when indexing a dictionary is a missing key error. But because the name `mydictionary` is misspelled, Python raises a `NameError` instead for the undefined name reference, which the handler will silently catch and ignore. Hence, the event handler will incorrectly fill in a `None` default for the dictionary access, masking the program error.

Moreover, catching `Exception` here will not help—it would have the exact same effect as an empty `except`, silently filling in a default and hiding an error you will probably want to know about. If this happens in code that is far removed from the place where the fetched values are used, it might make for an interesting debugging task!

As a rule of thumb, be as *specific* in your handlers as you can be—empty `except` clauses and `Exception` catchers are handy but potentially error-prone. In the last example, for instance, you would be better off listing `KeyError` in the `except` to avoid intercepting unrelated events. In simpler scripts, the potential for problems might not be significant enough to outweigh the convenience of a catchall, but in general, general handlers are generally trouble.



More closers: Python's `atexit` standard-library module allows programs to handle program shutdowns without recovery from them, and its `sys.excepthook` can be used to customize what the top-level exception handler does. A related call, `os._exit`, ends a program like `sys.exit`, but via immediate termination—it skips cleanup actions, including any registered with `atexit`, and cannot be intercepted with `try/except` or `try/finally`. It is usually used only in spawned child processes, a topic beyond this book's scope. See Python's library manual for more details.

Catching Too Little: Use Class-Based Categories

Being too specific in exception handlers can be just as perilous as being too general. When you list specific exceptions in a `try`, you catch only what you actually list. This isn't necessarily a bad thing, but if a system evolves to raise other exceptions in the future, you may need to go back and add them to exception lists elsewhere in your code.

We saw this phenomenon at work in the prior chapter. By way of review, because the following handler is written to treat only `MyExcept1` and `MyExcept2` as cases of interest, a future `MyExcept3` won't apply:

```
try:
    ...
except (MyExcept1, MyExcept2):    # Breaks if you add a MyExcept3 later
    ...
```

Careful use of class-based exceptions can make this code maintenance trap go away completely. By catching a general superclass, new exceptions don't imply `except`-clause changes:

```
try:
    ...
except CommonCategoryName:      # OK if you add a MyExcept3 subclass later
    ...
```

In other words, a little design goes a long way. The moral of the story is to be careful to be neither too general nor too specific in exception handlers and to pick the granularity of your `try` statement wrappings wisely. Especially in larger systems, exception policies should be a part of the overall design.

Core Language Wrap-Up

Congratulations! This concludes your voyage through the fundamentals of the Python programming language. If you’ve gotten this far, you’ve become a fully operational Python programmer. There’s more optional reading in the advanced topics part ahead described in a moment. In terms of the essentials, though, the Python story—and this book’s main journey—is now complete.

Along the way, you’ve seen just about everything there is to see in the language itself and in enough depth to apply to most of the code you are likely to encounter in the Python “wild.” You’ve studied built-in types, statements, and exceptions, as well as tools used to build up the larger program units of functions, modules, and classes.

You’ve also explored important software design issues, the complete OOP paradigm, functional programming tools, program architecture concepts, alternative tool trade-offs, and more—compiling a skill set now qualified to be turned loose on the task of developing real applications.

The Python Toolset

From this point forward, your future Python career will largely consist of becoming proficient with the toolset available for application-level Python programming. You’ll find this to be an ongoing task. The standard library, for example, contains hundreds of modules, and the public domain offers still more tools. It’s possible to spend decades seeking proficiency with all these tools—especially as new ones are constantly appearing to address new technologies.

Speaking generally, Python provides a hierarchy of toolsets:

Built-in tools

Built-in types like strings, lists, and dictionaries make it easy to write simple programs fast.

Python-coded extensions

For more demanding tasks, you can write your own functions, modules, and classes in Python itself.

Other-language extensions

Although we don’t cover this topic in this book, Python can also be extended with code written in an external language like C, C++, or Java.

Because Python layers its toolsets, you can decide how deeply your programs need to delve into this hierarchy for any given task—you can use built-ins for simple scripts, add Python-coded extensions for larger systems, and code other extensions for advanced work. We’ve only covered the first two of these categories in this book, and that’s plenty to get you started doing substantial programming in Python.

Beyond this, there are tools, resources, and precedents for using Python in nearly any computer domain you can imagine. For pointers on where to go next, see [Chapter 1](#)’s overview of Python applications and users. You’ll likely find that with a powerful open source language like Python, common tasks are often much easier, and even enjoyable, than you might expect.

Development Tools for Larger Projects

Most of the examples in this book have been fairly small and self-contained. They were written that way on purpose to help you master the basics. But now that you know all about the core language, it’s time to start learning how to use Python’s built-in and third-party interfaces to do real work.

In practice, Python programs can become substantially larger than the examples you’ve experimented with so far in this book. Even in Python, *thousands* of lines of code are not uncommon for nontrivial and useful programs once you add up all the individual modules in the system. Though Python’s basic program structuring tools, such as modules and classes, help much to manage this complexity, other tools can sometimes offer additional support.

For developing larger systems, you’ll find such support available in both Python and the public domain. You’ve seen some of these in action, and others have been noted in passing. This category morphs constantly, so we can’t get too detailed here, but to help you with your next steps, here is a quick tour and summary of tools in this domain:

Documentation tools

PyDoc’s help function and HTML interfaces were introduced in [Chapter 15](#). PyDoc provides a documentation system for your modules and objects, integrates with Python’s docstrings syntax, and is a standard part of the Python system. See [Chapters 15](#) and [4](#) for more documentation source hints.

Error-checking tools

Because Python is such a dynamic language, some programming errors are not reported until your program runs (even syntax errors are not caught until a file is run or imported). This isn’t a big drawback—as with most languages, it just means that you have to test your Python code before shipping it. With Python, you essentially trade a compile phase for an initial testing phase. Furthermore, Python’s dynamic nature, automatic error checking and reporting messages, and exception model make it easier and quicker to find and fix errors than it is in some other languages. Unlike C, for example, Python does not crash completely on errors.

Still, tools can help here too. As representative examples, the *PyChecker*, *Pylint*, and *Pyflakes* third-party systems provide support for catching common errors before your script runs. They serve similar roles to the *lint* program in C development. Some Python developers run their code through such tools prior to testing or delivery to catch any lurking potential problems. In fact, it’s not a bad idea to try this when you’re first starting out—some of these tools’ warnings may help you learn to spot and avoid common Python mistakes.

Testing tools

In [Chapter 25](#), we learned how to add self-test code to a Python file by using the `__name__ == '__main__'` trick at the bottom of the file—a simple unit-testing protocol. For more advanced testing purposes, Python comes with two testing tools. The first, *PyUnit* (called `unittest` in the standard-library manual), provides an object-oriented class framework for specifying and customizing test cases and expected results. It mimics the JUnit framework for Java and is a sophisticated class-based unit testing system.

The `doctest` standard-library module provides a second and simpler approach to regression testing based upon Python’s docstrings feature. Roughly, to use `doctest`, you cut and paste a log of an interactive testing session into the docstrings of your source files. `doctest` then extracts your docstrings, parses out the test cases and results, and reruns the tests to verify the expected results. See the library manual for more on both testing tools.

IDEs

We discussed IDEs for Python briefly in [Chapter 3](#). IDEs such as *PyCharm* and Python’s own *IDLE* provide a graphical environment for editing, running, debugging, and browsing your Python programs. Some advanced IDEs listed in [Chapter 3](#) may support additional development tasks, including source control integration, code refactoring, project management tools, and more. Though

aimed at roles other than general software development, *Jupyter notebooks* may qualify as a kind of IDE too. See [Chapter 3](#), the text editors page at *python.org*, and your favorite web search engine for more on available IDEs for Python.

Profilers

As we’ve seen, because Python is both dynamic and fluid, intuitions about performance gleaned from experience with other languages usually don’t apply to Python code. To truly isolate performance bottlenecks in your code and compare coding alternatives’ speed, you need to add timing logic with clock tools in the `time` or `timeit` modules or run your code under the `profile` module. We saw examples of the timing modules at work when comparing the speed of iteration tools and Pythons in [Chapter 21](#).

Profiling is often your first optimization step—code for clarity, then profile to isolate bottlenecks, and then time alternative codings of the slow parts of your program. For the second of these steps, `profile` and its optimized `cProfile` relative are standard-library modules that implement source code profiling for Python. After running code you provide, they print a report that gives performance statistics too detailed for us to cover here. See Python’s library manual for more on profilers, as well as the `pstats` module used to analyze results.

Debuggers

We discussed debugging options both in this part and in [Chapter 3](#) (see the latter’s sidebar “[Debugging Python Code](#)” on page 49). As a review, most development IDEs for Python support GUI-based debugging, and the Python standard library also includes a source code debugger module called `pdb`. This module provides a command-line interface and works much like common C language debuggers (e.g., *dbx*, *gdb*), and is detailed in Python’s library manual.

Because IDEs such as IDLE also include point-and-click debugging interfaces, `pdb` is more useful when a GUI isn’t available or when more control is desired. See [Chapter 3](#) for tips on using IDLE’s debugging GUI interfaces. As also noted in [Chapter 3](#), though, neither `pdb` nor IDEs seem to be used much in practice: most programmers simply either read Python’s error messages or insert `print` statements to add beacon displays and rerun—not the most high-tech of solutions, perhaps, but the practical tends to win the day in the Python world.

Shipping options

In “[Standalone Executables](#)” on page 24, we surveyed common tools for packaging Python programs. A variety of systems package program bytecode and the Python Virtual Machine into standalone executables, which don’t require that Python be installed on the host machine. In addition, we’ve learned that Python programs may be shipped in their source (*.py*) or bytecode (*.pyc*) forms, and *.zip* files may act like package folders. When your code is ready to go live as an open source tool, also see the web for resources on Python’s *pip* installer system.

Optimization options

When speed counts, there are numerous ways to optimize your Python programs, as enumerated in [Chapter 2](#). For instance, the *PyPy* system demoed in [Chapter 21](#) provides an automatic speed boost today, and others like *Shed Skin* and *Cython* offer different routes to faster programs. Although Python’s `-O` command-line flag noted in [Chapter 34](#) (and to be deployed in [Chapter 39](#)) optimizes bytecode, it yields a very modest performance boost, and is not commonly used except to remove debugging code and `asserts`.

Though a last resort, you can also move parts of your program to a compiled language such as C to boost performance; see Python’s manuals for more on C extensions. In addition, Python’s speed tends to improve over time, so upgrading to later releases may boost speed too—once you verify

that they are faster for your code, that is (though long since fixed, Python 3.X's early releases were radically slower than 2.X in some roles).

Installation management

If you need to install and segregate multiple sets of Python extensions on your machine, you may also wish to use *virtual environments*—noted briefly in [Chapter 22](#) and implemented by Python's standard-library module `venv`. This module allows you to create multiple virtual environments, each of which has its own independent set of Python packages, is contained in a directory, and is activated and deactivated by console commands. When a virtual environment is activated, tools such as `pip` install Python packages into that environment, and search paths are tailored for that environment's installs. See Python's library manual for more info.

Other hints for larger projects

We've also studied a variety of core-language topics in this text that may grow more useful once you start coding larger projects. These include module packages ([Chapter 24](#)), exceptions classes ([Chapter 34](#)), pseudoprivate class attributes ([Chapter 31](#)), documentation strings ([Chapter 15](#)), module data hiding ([Chapter 25](#)), and all the design and usage guidelines we've explored along the way for objects, statements, functions, modules, classes, and exceptions. If you've read this far, you're already well-equipped to level up.

To learn about these and many other larger-scale Python development tools, browse the PyPI website, python.org, and the web at large. Applying Python may be a larger topic than learning Python, but it is also one we'll have to delegate to follow-up resources here.

Chapter Summary

This chapter wrapped up the exceptions part of this book with a survey of design concepts, a look at common exception use cases, and a brief summary of commonly used development tools.

This chapter also wrapped up the core material of this book. At this point, you've been exposed to the full subset of Python that most programmers use—and probably much more. In fact, by virtue of reaching these words, you should feel free to consider yourself an *official Python programmer*. Be sure to pick up a t-shirt or laptop sticker the next time you're online (and don't forget to add Python to your résumé the next time you dig it out).

The next and final part of this book is a collection of chapters dealing with topics that are advanced but still in the core-language category. These chapters are all *optional reading*, or at least *deferrable reading*, because not every Python programmer must delve into their subjects, and others can postpone these chapters' topics until they are needed. Indeed, many of you can stop here and begin exploring Python's roles in your application domains. Frankly, application libraries tend to be more important in practice than advanced—and, to some, esoteric—language features.

On the other hand, if you do need to care about things like Unicode or binary data (and you probably do!); have to deal with API-building tools such as descriptors, decorators, and metaclasses; or just want to dig a bit further in general, the next part of the book will help you get started. The larger examples in the final part will also give you a chance to see the concepts you've already learned being applied in more realistic ways.

As this is the end of the core material of this book, though, you get a break on the chapter quiz—just one question this time. As always, be sure to work through this part's closing exercises to cement what you've learned in the past few chapters; because the next part is optional reading, this is the final end-of-part exercises session. If you want to see some examples of how what you've learned comes together

in real scripts drawn from common applications, be sure to check out the “solution” to this part’s exercise 4 in [Appendix B](#).

And if this is where you’ll be disembarking from this book’s voyage, be sure to also see “[Encore: Print Your Own Completion Certificate!](#)” on [page 1128](#) at the end of [Chapter 41](#), the very last chapter in this book (for the sake of readers continuing on to the Advanced Topics part, this chapter won’t spill the beans here).

Test Your Knowledge: Quiz

1. What’s up with the mouse on the cover of this book?

Test Your Knowledge: Answers

1. OK, this was never mentioned and is hardly a fair question, but for the record: the mouse—really, a wood rat, *Neotoma muridae*—was chosen for this book’s first edition by its publishing company in the 1990s, based on the fact that this animal is common food for a python. The idea was that the wood rat must learn about the python to avoid being eaten by it. Clever, to be sure, but this also came with a subtler tie-in about *Neotoma* being pack rats attracted to shiny objects that compulsively collect whatever they come across, which seems an apt metaphor for Python’s history of language-feature accumulation.

So enjoy the shiny objects, but don’t get eaten by the constricting reptiles along the way.

Test Your Knowledge: Part VII Exercises

As we’ve reached the end of this part of the book, it’s time for a few exception exercises to give you a chance to practice the basics. Exceptions really are simple tools; if you’re able to work through these exercises, you’ve probably mastered the exceptions domain. See “[Part VII, Exceptions](#)” on [page 1188](#) in [Appendix B](#) for the solutions.

1. **try/except:** Write a function called `oops` that explicitly raises an `IndexError` exception when called. Then, write another function that calls `oops` inside a `try/except` statement to catch the error. What happens if you change `oops` to raise a `KeyError` instead of an `IndexError`? Where do the names `KeyError` and `IndexError` come from? (Hint: recall that all unqualified names generally come from one of four scopes.)
2. **Exception objects and lists:** Change the `oops` function you just wrote to raise an exception you define yourself, called `MyError`. Identify your exception with a class of your own. Then, extend the `try` statement in the catcher function to catch this exception and its instance in addition to `IndexError`, and print the instance you catch.
3. **Error handling:** Write a function called `safe(func, *pargs, **kargs)` that runs any function with any number of positional and/or keyword arguments by using the `*` arbitrary arguments header and call syntax, catches any exception raised while the function runs, and prints the exception using the `exc_info` call in the `sys` module. Then use your `safe` function to run your `oops` function from exercise 1 or 2. Put `safe` in a module file called `exctools.py`, and pass it the `oops` function interactively. What kind of error messages do you get? Finally, expand `safe` to also print a Python stack trace when an error occurs by calling the built-in `print_exc` function in the standard-library `traceback` module; see earlier in this chapter, and consult the Python library reference manual for

usage details. We could probably code `safe` as a *function decorator* per the Chapter 19 and 32 introductions, but we'll have to move on to the next part of the book to learn fully how (see the solutions for a preview).

4. *Self-study examples*: At the end of [Appendix B](#) in “[Part VII, Exceptions](#)” on [page 1188](#), this book lists a handful of example scripts developed as group exercises in live Python classes for you to study on your own in conjunction with Python's standard manual set. These are not described, and they use tools in the Python standard library that you'll have to research yourself. Still, for many readers, it helps to see how the concepts we've discussed in this book come together in real programs. If these pique your interest for more, you can find a wealth of larger and more realistic application-level Python program examples in follow-up books and on the web: pick your domain, and start exploring!

Advanced Topics

Unicode and Byte Strings

In [Chapter 7](#), the Python string story was watered down on purpose to help you get started with the fundamentals. Now that you’ve learned the basics, this chapter moves on to extend them to include the full Unicode-text and binary-data string tales in Python.

This extension was more optional in earlier editions of this book because Unicode was an afterthought in Python 2.X. Python 3.X elevates it to required reading because its normal strings simply *are* Unicode. Still, how much you need to care about this topic depends in large part upon which of the following categories you fall into:

- If you deal with non-ASCII *Unicode text*—for instance, in the context of internet content, internationalized applications, XML parsers, and some GUIs—you will find direct and seamless support for text encodings in both Python’s all-Unicode `str` object, as well as its Unicode-aware text files.
- If you deal with *binary data*—for example, in the form of image or audio files, network transfers, or packed data shared with lower-level tools—you will need to understand Python’s bytes object and its sharp distinction between text and binary data and files.
- If you fall into *neither* of the prior two categories, you may be able to defer this topic and use strings as you did in [Chapter 7](#): with the general `str` object, text files, and all the familiar string operations. Your strings will be encoded and decoded using your platform’s default Unicode encoding, but you won’t notice—until, as you’ll see, you encounter content or platforms that use a different default!

To be sure, if text is always ASCII in your corner of the software world, you might be able to get by with simple string objects and text files and can avoid much of the story that follows. As you’ll learn in a moment, ASCII is a simple kind of Unicode and a subset of other common encodings, so string operations and files “just work” if your programs process ASCII text only and will never deviate from this limitation.

Even if this chapter’s topics seems remote to you today, though, a basic understanding of Python’s string model can both demystify some of the underlying details now and prepare you for Unicode or binary-data issues that may impact you in the future. Given the prominence of the web in most software careers today, that impact may be more a matter of *when* than *if*.

Unicode Foundations

Before jumping into code, let's begin with a general overview of the Unicode model and Python's support for it. To fully understand both, we have to start with a brief look at how characters are actually represented in computers.

Character Representations

Most programmers think of strings as a series of characters (really, their integer codes) used to represent textual data. That's still true in the brave new world of Unicode, but the way characters are stored in a computer's memory and files can vary, depending on both what sort of characters are recorded and how programmers choose to record them.

For many programmers in the US, *ASCII* formed their original notion of text strings. ASCII is a standard that defines character codes 0...127 (which always means an inclusive range in this chapter) and thus allows each character to be stored in one 8-bit byte (using 7 bits). For example, the ASCII standard maps character a to the integer value 97 (0x61 in hex), which can be stored in a single byte both in computer memory and on files.

To witness this for yourself, Python's built-in `ord` function shows the integer code of a given character; `chr` reveals the character of a given integer code; and `hex` gives the code's byte value as two hex digits, each of which fits a 4-bit “nibble.” The first of these, `ord`, is the value of a character's representation code—and byte—in ASCII:

```
$ python3          # Or py -3 on Windows
>>> ord('a')      # Character => code
97
>>> chr(97)       # Code => character
'a'
>>> hex(97)       # Byte value: fits 8 bits
'0x61'
>>> 0b0111_1111   # Limit of ASCII's 7-bit range
127
```

ASCII makes text processing simple, because characters directly correlate to bytes. Sometimes, though, this isn't enough. Accented characters and special symbols, for example, do not fit into the range of character codes defined by ASCII. To allow for some such extra characters, other standards allow all possible values in an 8-bit byte, 0...255, to be used as codes, and assign values 128...255 to additional characters.

One such standard is known as *Latin-1* and is widely used in Western Europe. In Latin-1, character codes above 127 are assigned to accented and otherwise special characters. For instance, the character that Latin-1 assigns to code 196 (a.k.a. byte value 0xc4) is a specially marked and non-ASCII character, Ä. In Python:

```
>>> chr(196)      # Too big for ASCII
'Ä'
>>> ord('Ä')      # Okay for Latin-1
196
>>> hex(ord('Ä'))  # Byte value in Latin-1
'0xc4'
>>> bin(ord('Ä'))  # Latin-1 uses all 8 bits
'0b11000100'
```


Still, some alphabets define so many characters that it is impossible to represent them as one byte-sized code per character. The integer codes of the symbols and characters in the following, for example, require more space than a byte—as do those of all the emojis that may not work in some tools but manage to crop up in your emails and texts anyhow:

```
>>> ord('€')
9758
>>> hex(ord('€'))           # Too big for one byte
'0x261e'

>>> [hex(ord(c)) for c in '真几👉']   # Ditto: Unicode required
['0x771f', '0x41b', '0x21e8']

>>> [hex(ord(c)) for c in '😄😄👍']   # Emojis: > two bytes (16 bits)
['0x1f642', '0x1f64a', '0x1f44d']
```

Unicode provides the generality we need to deal with text containing non-ASCII characters and symbols like these. In fact, it defines and assigns enough *character codes* to represent almost every natural language in use, plus a large set of symbols and emojis. In Unicode speak, these codes that stand for characters take the form of numbers (integers), and are usually called *code points*. The code points that Unicode assigns to characters a, Ä, and 😄, for instance, are 97, 196, and 128578 (0x61, 0xc4, and 0x1f642 in hex), respectively:

```
>>> [f'{c} is {ord(c)} and {hex(ord(c))}' for c in 'aÄ😄']
['a is 97 and 0x61', 'Ä is 196 and 0xc4', '😄 is 128578 and 0x1f642']
```

Unicode is sometimes referred to as “wide-character” strings because its range of characters is so broad that multiple bytes may be needed to represent individual character codes. Such text is readily stored in computer memory because each character code can simply span as many bytes as its code-point number requires (the exact way this is done can vary by programming language and isn’t consequential to your Python code).

Once text leaves your computer, though, its storage is more constrained: bytes are a bad thing to waste on your drives and networks, and text used across platforms must follow the same formatting rules. To allow for this, Unicode also defines standard ways to map character codes to and from bytes for storage and transmission that are both platform- and language-neutral—the *encodings* we’ll explore in the next section.

The takeaway here is that Unicode’s combination of all-encompassing character codes and their predefined encodings make it a portable and flexible model, and the standard way that programs deal with non-English and other text that may have more characters than 8-bit bytes can handle. As an added bonus, earlier schemes like ASCII also fall under the Unicode umbrella unchanged, but we have to move on to the next section to see how.

Character Encodings

One of the keys to understanding how Unicode works lies in the way its integer character codes (a.k.a. code points) that represent characters are mapped to their encoded forms for efficient storage or transfer. Code points in memory are just integers of arbitrary size, but storage and transfer, by nature, impose constraints on time, space, and interoperability that warrant extra formatting steps.

In the Unicode world, we say that characters are translated to and from raw bytes using an *encoding*—the rules for translating a Unicode-text string into a sequence of bytes and extracting the same string from its sequence of bytes. More procedurally, this translation back and forth between bytes and strings is defined by two terms (the first of which doubles as a noun and verb, confusingly!):

- *Encoding* is the process of translating a string of characters into its raw-bytes form, per any desired encoding that's broad enough to store the string's characters.
- *Decoding* is the process of translating a string of raw bytes into its character-string form, per the encoding originally used to create the bytes string.

As we've seen, Unicode defines both character codes and a set of standard encodings. For some of the encodings it defines, the translation process is trivial—ASCII and Latin-1, for instance, map each character to a single byte, so little or no work is required to encode and decode if characters are the same bytes in memory too.

You can view this for yourself with the `encode` method available on all Python text strings, which simply returns the bytes used to encode the string. The following means that the ASCII character `a` occupies just one byte when encoded per the ASCII encoding:

```
>>> len('a'.encode('ASCII'))      # ASCII 'a' encodes in 1 byte per ASCII
1
```

For other encodings, the mapping can be more complex and yield multiple bytes per character. The widely used *UTF-8* encoding, for example, allows more characters to be represented by employing a variable-number-of-bytes scheme that's both general and economical. In fact, because UTF-8 can handle any Unicode code point, it's become a de facto standard of sorts for text.

In UTF-8, character codes less than 128 are represented as a single byte; codes between 128 and 0x7ff (2047) are turned into two bytes, where each byte has a value between 128 and 255; and codes above 0x7ff are turned into three- or four-byte sequences having values between 128 and 255. This keeps simple ASCII strings compact, sidesteps byte ordering issues, and avoids null (zero) bytes that can cause problems for C libraries and networking. In Python:

```
>>> len('a'.encode('UTF-8'))      # ASCII: encodes in 1 byte
1
>>> len('Ä'.encode('UTF-8'))      # Non-ASCII: encodes in 2 bytes
2
>>> len('🍌'.encode('UTF-8'))      # Emoji: encodes in 4 bytes
4
```

Despite such details, it's important to note that ASCII is a *subset* of both Latin-1 and UTF-8. This is true because these encodings encode ASCII characters to bytes the same way as ASCII. That, in turn, makes these encodings backward compatible with existing ASCII data: every character string encoded per ASCII is also valid according to the Latin-1 and UTF-8 encodings, and every ASCII file is a valid Latin-1 and UTF-8 file.

Technically, the ASCII encoding is a 7-bit subset of the other two: it's binary compatible with all character codes less than 128. Latin-1 and UTF-8 simply allow for additional characters: Latin-1 for characters mapped to values 128...255 within a byte, and UTF-8 for characters that may be represented with multiple bytes. The converse is not true, however: UTF-8 and Latin-1 text is not compatible with the ASCII encoding unless its text's code-point values are all less than 128; otherwise, encoding or decoding per ASCII fails.

In Python again, the mapping is easy to observe. Per the following, an ASCII character encodes to the same single byte in ASCII, UTF-8, and Latin-1 encodings, but non-ASCII characters do not, and require more general encodings than ASCII to encode at all (the `b'...'` here is the Python bytes object, which will soon play a leading role in this chapter):

```
>>> 'a'.encode('ASCII')           # ASCII encodes to the same byte
b'a'
>>> 'a'.encode('UTF-8')
b'a'
>>> 'a'.encode('Latin-1')
b'a'

>>> 'Ä'.encode('UTF-8')           # But non-ASCII requires more bytes
b'\xc3\x84'
>>> '🐶'.encode('UTF-8')          # And more inclusive encodings
b'\xf0\x9f\x99\x82'

>>> '🐶'.encode('ASCII')
UnicodeEncodeError: 'ascii' codec can't encode character '\U0001f642'...
>>> '🐶'.encode('Latin-1')
UnicodeEncodeError: 'latin-1' codec can't encode character '\U0001f642'...
```

Other encodings support richer character sets in other ways. For instance, *UTF-16* and *UTF-32* use a fixed and larger 2 and 4 bytes per character, respectively, the former with a special *surrogate-pair* protocol for codes too large for 2 bytes. Both of these, along with UTF-8, may also allow or require a *BOM* (Byte Order Marker) preamble at the start of encoded text, which can designate byte order and encoding type, may be present in encoded text stored in files or memory, and is automatically handled for text-mode files.

We'll skip further details here for space (watch for the BOM to drop at the end of this chapter, along with the thorny topic of Unicode normalization), but keep in mind that all of these—ASCII, Latin-1, UTF-8, and others—are simply alternative Unicode encodings that yield the same Unicode code-point text when decoded. The net effect ensures that text is *portable* across all the tools that use it in exchange for minor translation costs:

- When *decoded*, character code points may or may not occupy multiple bytes in memory, depending on programming-language implementation. Some recent Pythons, for example, use a variable-length scheme to store decoded text with 1, 2, or 4 bytes per character, depending on string content. Earlier Pythons instead store each character in a fixed 2 or 4 bytes, depending on compilation settings.
- When *encoded*, the format of character code points is wholly determined by the standard Unicode encoding applied. This format is the same regardless of which programming language creates or processes the text, making it ideal for storage and transfer—especially in the diverse realm of the internet. This format is often less ideal for programs to use, though, which is why it's normally decoded when loaded.

To Python programmers, an encoding is specified as a string containing the encoding's name. Python comes with roughly 100 different encodings out of the box; see the `codecs` module in the Python Library Reference for a list. Importing module `encodings` and asking for `help(encodings)` shows you many as well. Some encodings are implemented in Python, and some in C, and many have multiple names; for example, `latin-1`, `iso_8859_1`, and `8859` are all synonyms for the same encoding, `Latin-1`. We'll revisit encodings later in this chapter when we study Unicode coding techniques.

For another take on the Unicode backstory, see the Python standard manual at python.org. It includes a [Unicode HOWTO section](#), which provides additional minutiae that we will skip here to focus on the fundamentals.

Introducing Python String Tools

At a more concrete level, the Python language provides multiple string data types to represent content in your script: both *textual data*—integer code-point values of decoded Unicode characters in memory—as well as *binary data*—raw byte values, including text that is in encoded form. All told, Python comes with three string object types:

- `str`—for representing Unicode text (decoded code points)
- `bytes`—for representing binary data (including encoded text)
- `bytearray`—a mutable flavor of the `bytes` type

All three types support similar operation sets but have very different roles and cannot generally be mixed because of this. Moreover, files and other content tools reflect the text/binary dichotomy, too, and use specific string types in different nodes. The next sections introduce the salient points of this model.

The `str` Object

First up, the basic `str` type (e.g., `'text'`) is for decoded Unicode text. It's formally defined as an *immutable sequence of characters*—which means code points that are not necessarily bytes. Its content may contain both simple text, such as ASCII, whose encoded and decoded forms might yield one byte per character, as well as richer Unicode text, whose encoded and decoded forms may both require multiple bytes per character.

In memory, a `str` is just an ordered collection of Unicode code-point integers, which print as *glyphs*—visual representations that may vary from host to host—of the characters that the code points represent. When transferred to and from files, a `str` is automatically encoded to and decoded from a sequence of bytes using either the host platform's default or a provided encoding name to translate with an explicit scheme. `str` objects themselves, however, have *no notion of an encoding*; they are just character code points.

The `bytes` Object

While `str` is great for Unicode text, many programs need to process raw binary content that is not encoded per any Unicode format—as well as the bytes used to store text when it is encoded. Image files and packed data you might process with Python's `struct` module fall into this category. To accommodate this, the `bytes` type supports processing of truly binary data. `bytes` is just raw bytes, not Unicode-text characters, though its content may include the bytes of still-encoded text, which always has an implied encoding.

The `bytes` type is formally defined as an *immutable sequence of 8-bit integers*. Its content represents byte values, and it supports almost all the same operations that the `str` type does; this includes string methods, sequence operations, and even `re` module pattern matching (formatting works on `bytes` today, too, but was added later in 3.X's evolution).

A `bytes` object really is a sequence of small integers, each of which is in the range 0...255: indexing a `bytes` returns an `int`, slicing one returns another `bytes`, and running `list` on one returns a list of integers, not characters. However, when processed with operations that assume characters (e.g., the `isalpha` method), the contents of `bytes` objects are assumed to be ASCII-encoded bytes. Further, `bytes` items whose values fall in the range of ASCII character codes are printed as ASCII-character glyphs instead of integers or their hex escapes; this is done for convenience, though it may also confuse the distinction between text and binary data.

The bytearray Object

Though less commonly used, Python also comes with `bytearray`, a variant of `bytes` that is *mutable* and so supports in-place changes. The `bytearray` type provides the usual string operations that `str` and `bytes` do but also has many of the same in-place change operations as lists (e.g., `append` and `extend` methods and assignment to indexes). Assuming your strings can be treated as raw bytes, `bytearray` adds direct in-place mutability for string data—something long prohibited by `str` and `bytes`.

Text and Binary Files

Because file I/O is one of the main benefactors of encodings, it's also a core Unicode tool. As we've seen, text is really just decoded integer character codes when it is in memory; it's when text is transferred to and from *external interfaces* like files that Unicode encodings come into play. By contrast, truly binary data may have nothing at all to do with encodings—or text at all. Because of this, Python makes a sharp platform-independent distinction between text and binary files accessed with the built-in `open` function:

- When a file is opened in *text mode*, reading its data automatically decodes its content and returns it as a `str`, and writing takes a `str` and automatically encodes it before transferring it to the file. In both cases, the encoding to use is either a platform default or a provided `encoding` argument to `open`. Text mode files also support universal newline (a.k.a. end-of-line) translation, BOMs, and other encoding arguments.
- When a file is opened in *binary mode* by adding a `b` to the mode string argument in the `open` call, reading its data does not decode it in any way and simply returns its content raw and unchanged as a `bytes` object. Writing similarly takes a `bytes` object and transfers it to the file unchanged, and binary-mode files also accept a `bytearray` object for the content to be written to the file.

Because `str` and `bytes` are sharply differentiated by the language this way, you must decide whether your data is text or binary in nature and use `str` or `bytes` objects to represent its content in your script, respectively. Ultimately, the mode in which you open a file will dictate which type of *object* your script will use to represent its content:

- If you are processing image or audio files, packed data created by other programs whose content you must extract, and some device data streams, chances are good that you will want to deal with it using `bytes` and binary-mode files. You might also opt for `bytearray` to update the data without making copies of it in memory.
- If instead you are processing something that is textual in nature, such as program output, HTML or JSON content, and CVS or XML files, you probably want to use `str` and text-mode files.

Subtly, the mode-string argument to `open` (its `mode` keyword and second positional argument) becomes fairly crucial—its content not only specifies a file processing *mode* but also implies a Python object *type*. By adding a `b` (lowercase only) to the mode string, you specify a binary mode file and will receive, or usually provide, a `bytes` object to represent the file's content when reading or writing. Without the `b`, your file is processed in text mode, and you'll use `str` objects to represent its content. For example, modes `rb`, `wb`, and `rb+` imply bytes, but `r`, `w+`, and `rt` (the default: read text) imply `str`.

If you're anxious to see files in action, watch for the examples ahead, especially those of Unicode text files. To understand file usage in full, though, we first need to explore string operations as they extend to Unicode and bytes.

Using Text Strings

Let's step through a few examples that demo the prior section's string types live. Here, our primary focus is on using these types for text (we'll explore binary roles later). Along the way, you'll see how literals, conversions, and non-ASCII text are coded in Python.

Literals and Basic Properties

Most Python string objects are born when you call a built-in function such as `str` or `bytes`, process a file created by calling `open`, or code literal syntax in your script. For the latter, the usual `'...'` makes a `str`; a unique literal form `b'...'` is used to create a `bytes`; and `bytearray` objects may be made by calling the same-named function with a variety of possible arguments.

More formally, all the usual string literal forms we met in [Chapter 7](#)—`'...'`, `"..."`, and triple-quoted blocks—generate a `str`; adding a `b` or `B` just before them creates a `bytes` instead. This `b'...'` (and equivalently, `B'...'`) bytes literal is similar in spirit to the `r'...'` raw string we've also met, which suppresses backslash escapes; in fact, the two prefixes can be combined to use backslashes verbatim in a `bytes`. Consider the following:

```
>>> B = b'code'           # Make a bytes object: 8-bit bytes
>>> S = 'hack'            # Make a str object: Unicode characters

>>> type(B), type(S)
(<class 'bytes'>, <class 'str'>)

>>> B                      # Sequence of ints, prints as ASCII characters
b'code'
>>> S                      # Sequence of code points, prints as text glyphs
'hack'

>>> B2 = B""              # bytes prefix works on single, double, triple
xxxx
yyyy
""""

>>> B2
b'\nxxx\nyyyy\n'

>>> b'A\nB\rC', br'A\nB\rC', rb'A\nB\rC'    # Raw-string combos work too
(b'A\nB\rC', b'A\\nB\\rC', b'A\\nB\\rC')
```

Once you have a string, all the usual operations we met earlier work, but their results are type specific (bytes is integers and str is characters), and immutability still applies:

```
>>> B, S
(b'code', 'hack')

>>> B[0], S[0]                # Indexing returns an int for bytes, str for str
(99, 'h')

>>> B[1:], S[1:]              # Slicing makes another bytes or str
(b'ode', 'ack')

>>> list(B), list(S)
([99, 111, 100, 101], ['h', 'a', 'c', 'k'])    # bytes is really ints

>>> B[0] = 'x'                 # Both are immutable
TypeError: 'bytes' object does not support item assignment
>>> S[0] = 'x'
TypeError: 'str' object does not support item assignment
```

Because they apply to more than text and have some unique behaviors, we'll defer `bytearray` and more about bytes to a dedicated section later in this chapter.



Blast from the past: Python 3.X also recognizes Python 2.X's Unicode string literals to ease migration of 2.X code: a 2.X `u'...'` literal in Python 3.X is just a synonym for a 3.X `'...'` str literal. This makes sense, given that 3.X's str is all Unicode, and allows some 3.X code to run on 2.X and vice versa. In today's 3.X world, though, there's no compelling reason to use the `u'...'` literal anymore (unless you're a fan of superfluous prefixes), but it may crop up in Python code you'll encounter in the wild. 2.X had a very long shelf life, after all.

String Type Conversions

Syntax aside, the first thing you might notice about Python strings is what they *cannot* do—str and bytes never mix automatically in expressions and generally are not converted to one another automatically when passed to functions. A function that expects an argument to be a str may not accept a bytes (and vice versa), and operators are fully rigid:

```
>>> 'hack' + b'code'
TypeError: can only concatenate str (not "bytes") to str
```

This is easier to understand if you remember that a text string may be radically different in its encoded and decoded forms, and Python has no idea what the content of a bytes is: if the bytes is encoded text, its encoding is unknown, but it may also be binary data (e.g., a loaded audio file) that has nothing to do with text at all.

Because of this ambiguity, Python basically requires that you either commit to one type or the other or perform manual, explicit conversions with the following tools (where ? means optional):

- S.encode(encoding?) and bytes(S, encoding)
Encode a str object *S* to a new bytes per *encoding*
- B.decode(encoding?) and str(B, encoding)
Decode a bytes object *B* to a new str per *encoding*

Both the preceding `S.encode()` and `B.decode()` methods and the file open call we'll explore ahead use either an explicitly passed-in encoding name or a default. The methods' default is always UTF-8 (by contrast, `open` uses a value in the `locale` module you'll meet shortly that may vary per platform, settings, and run and should usually be avoided):

```
>>> S = 'hack'
>>> S.encode()                # str to bytes: encode text into raw bytes
b'hack'

>>> bytes(S, encoding='ascii') # str to bytes, alternative
b'hack'

>>> B = b'code'
>>> B.decode()                # bytes to str: decode raw bytes into text
'code'

>>> str(B, encoding='ascii')   # bytes to str, alternative
'code'
```

Putting this together solves our original type error and allows us to mix strings and bytes as either encoded or decoded text:

```
>>> S, B
('hack', b'code')

>>> S.encode('ascii') + B      # bytes + bytes (encoded)
b'hackcode'

>>> S + B.decode('ascii')      # str + str (code points)
'hackcode'
```

A few cautions on defaults here. First of all, the encoding argument to `bytes` is *not optional*, even though it is in `S.encode()` (and `B.decode()`). More subtly, although `str` does not require the encoding argument like `bytes` does, leaving it off in `str` calls does not mean it defaults—instead, due to Python history, a `str` without an encoding returns the `bytes` object's *print string*, not its decoded and converted `str` form (this is usually not what you'll want!).

Assuming again that `B` and `S` are still as in the prior listing:

```
>>> bytes(S)
TypeError: string argument without an encoding

>>> str(B)                # str() works without encoding
'b'code'                  # But print string, not conversion!
>>> len(str(B))
7

>>> len(str(B, encoding='ascii')) # Pass encoding to convert to str
4
```

Also in the defaults department, your platform's various default encodings are available in the `sys` and `locale` modules but aren't as trustworthy as you might think:


```
$ py -3
>>> import sys, locale
>>> sys.platform                                # Underlying platform: Windows
'win32'
>>> sys.getdefaultencoding()                    # Methods default (but not for str(!))
'utf-8'
>>> locale.getpreferredencoding(False)          # open() default: a Latin-1 superset
'cp1252'
```

As shown, the `open` function's default file encoding lives in the `locale` module. On the PC used for Windows examples in this chapter, it's `cp1252`—a superset of Latin-1 that adds characters like slanted quotes. Defaults may differ, however, on other *platforms* and technically can even depend on environment-variable settings, command-line arguments, and settings on individual host machines.

For example, here's the differing case on this chapter's macOS host—like most Unix platforms, its `open` defaults to *UTF-8*:

```
$ python3
>>> import sys, locale
>>> sys.platform                                # Underlying platform: macOS
'darwin'
>>> sys.getdefaultencoding()                    # The default for methods is same
'utf-8'
>>> locale.getpreferredencoding(False)          # But this differs on Unix: be explicit!
'utf-8'
```

Besides such program-host differences, keep in mind that text *content* you receive from disparate sources might also use any Unicode encoding at all, making your host's default a moot point. Hence, your programs shouldn't generally rely on `open` defaults if they may need to care about portability now or in the future—always pass an explicit encoding to `open` when interoperability counts. We'll revisit encoding defaults and learn how to provide an explicit encoding to `open` when we explore Unicode files later in this chapter.

Having said all that, it's important to also note that encoding and decoding are substantially more than simple programming-language type conversions; really, they produce very different kinds of data. Encoding returns the bytes that result from transforming a text string per a Unicode scheme, and decoding returns the text string that is produced by undoing that transformation. While this is a conversion of sorts, and the mapping may seem trivial for simple text like ASCII, Unicode tends to make much more sense if you avoid blurring the distinction—especially for richer types of text like that in the next section.

Coding Unicode Strings in Python

Encoding and decoding grow more meaningful when you start dealing with non-ASCII Unicode text. To code Unicode characters that may be difficult to type on your keyboard, Python string literals support both:

- `\xNN` hex escapes, where two hex digits (*NN*) specify a character code as a 1-byte (8-bit) numeric value
- `\unNNNN` and `\UNNNNNNNNN` Unicode escapes, where the first *lowercase* form gives 4 hex digits to denote a 2-byte (16-bit) character code, and the second *uppercase* form gives 8 hex digits for a 4-byte (32-bit) code

Importantly, in `str` objects, all three of these escapes are used to give a Unicode character's *code-point* value—not its encoded bytes. By contrast, `bytes` objects allow only hex escapes for byte values; for text, this gives its *encoded form*—not its decoded code points.

Let's see how this all translates to code. Simple 7-bit *ASCII* text is formatted with one character per byte under most of the encoding schemes described near the start of this chapter (again, this is why *ASCII* passes as a binary-compatible subset of many other schemes):

```
>>> ord('X')           # Character 'X' has code-point value 88
88
>>> chr(88)            # Code-point 88 stands for character 'X'
'X'

>>> S = 'XYZ'          # str: code points display as their character glyphs
>>> S
'XYZ'
>>> len(S)              # 3 characters (not necessarily bytes) long
3

>>> S.encode('ascii')   # Values 0...127 in 1 byte each (ASCII shown as chars)
b'XYZ'
>>> S.encode('latin-1') # Values 0...255 in 1 byte each
b'XYZ'
>>> S.encode('utf-8')    # Values 0...127 in 1 byte, 128...2047 in 2, others 3~4
b'XYZ'
```

By contrast, the less common *UTF-16* and *UTF-32* use 2 and 4 bytes for every character, respectively, even for simple text like *ASCII*. This makes these encodings' data fast to process but may consume extra space and bandwidth, which renders them subpar in some applications. In the following, *ASCII* bytes print as characters, non-*ASCII*s print as `\xNN` escapes, padding bytes follow text, and each result has a 2- or 4-byte BOM header at the front whose details we're largely ignoring here (again, stay tuned for more on BOMs near the end of this chapter):

```
>>> S
'XYZ'

>>> S.encode('utf-16')   # Always 2 or 4 bytes per character, with BOM header
b'\xff\xfe\x00X\x00Y\x00Z\x00'

>>> S.encode('utf-32')
b'\xff\xfe\x00\x00X\x00\x00Y\x00\x00Z\x00\x00\x00'
```

To code *non-ASCII* characters, you can use hex and Unicode escapes in your strings. The numeric values coded as hexadecimal literals `0xC4` and `0xE8`, for instance, are the Unicode code points used to represent two special characters outside the 7-bit range of *ASCII*; we can embed them in `str` objects anyhow because `str` supports Unicode in full:

```
>>> chr(0xc4)           # 0xC4 and 0xE8 are accented characters outside ASCII
'Ä'
>>> chr(0xe8)
'è'

>>> S = '\xc4\xe8'      # Hex escapes: code-point values, not encoded bytes
>>> S
'Äè'

>>> S = '\u00c4\u00e8'  # Unicode escapes: 16-bits (2-bytes)
>>> S
'Äè'
```

```
>>> len(S)                # 2 characters long (not number of bytes!)
2
```

Now, if we try to encode a non-ASCII string like this to raw bytes as ASCII, we'll get an error. Encoding as Latin-1 works, though, and allocates 1 byte per character; encoding as UTF-8 allocates 2 bytes per character instead. If you write this string to a text-mode file, the raw bytes shown are what is actually stored on the file for the encoding types given:

```
>>> S = '\u00c4\u00e8'
>>> S.encode('ascii')
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
ordinal not in range(128)

>>> S.encode('latin-1')    # 1 byte per character
b'\xc4\xe8'

>>> S.encode('utf-8')      # 2 bytes per character
b'\xc3\x84\xc3\xa8'

>>> len(S.encode('latin-1')) # 2 bytes in latin-1, 4 in utf-8
2
>>> len(S.encode('utf-8'))
4
```

You can also go the other way—from raw bytes back to a Unicode string. You could read raw bytes from a file and decode manually this way, but the encoding mode you give to the open call causes this decoding to be done for you automatically (and avoids issues that may arise from reading partial character sequences when reading by blocks of bytes):

```
>>> B = b'\xc4\xe8'
>>> B
b'\xc4\xe8'
>>> len(B)                # 2 raw bytes, 2 characters
2
>>> B.decode('latin-1')   # Decode to latin-1 text
'Äè'

>>> B = b'\xc3\x84\xc3\xa8'
>>> len(B)                # 4 raw bytes
4
>>> B.decode('utf-8')
'Äè'
>>> len(B.decode('utf-8')) # 2 Unicode characters
2
```

When needed, you can also specify both 16- and 32-bit Unicode code-point values for characters in your str strings: use \u... with 4 hex digits for the former and \U... with 8 hex digits for the latter. As the last example in the following shows, you can also build such strings up piecemeal using chr, but it might become tedious for large strings:

```
>>> S = 'A\u00c4B\u0000\u0000e8C'
>>> S                # A, B, C, and 2 non-ASCII characters
'AÄBèC'
>>> len(S)           # 5 characters long
5

>>> S.encode('latin-1')
b'A\xc4B\xe8C'
>>> len(S.encode('latin-1')) # 5 bytes in latin-1
5
```

```

>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'
>>> len(S.encode('utf-8'))           # 7 bytes in utf-8
7

>>> S.encode('cp500')                 # Two other Western European encodings
b'\xc1c\xc2T\xc3'
>>> S.encode('cp850')                 # 5 bytes each
b'A\x8eB\x8aC'

>>> S = 'code'                        # ASCII text is the same in most
>>> S.encode('latin-1')
b'code'
>>> S.encode('utf-8')
b'code'
>>> S.encode('cp500')                 # But not in cp500: IBM ebcdic
b'\x83\x96\x84\x85'
>>> S.encode('cp850')
b'code'

>>> S = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'   # str the hard way
>>> S
'AÄBèC'

```

Another distinction to keep in mind: Python allows special characters' code points to be coded with both hex and Unicode escapes in `str` string literals but allows only *hex* escapes in bytes literals—and prints a warning in its recent versions if you violate this rule. In fact, Unicode escape sequences are taken *verbatim* in bytes and not as escapes.

This makes sense if you remember that bytes objects hold binary data, both textual and not. When they contain text, they hold characters' *encoded* bytes—not their decoded code points. Thus, Unicode code-point escapes simply don't apply to bytes, and hex escapes in their literals yield raw byte values, not characters.

This is true even though code-point and encoded-byte values happen to be the same for some characters in some encodings (confusingly!). Because bytes are not code points, they also must be decoded to `str` to print their non-ASCII characters properly:

```

>>> S = 'A\xc4B\xe8C'                # str recognizes hex and Unicode escapes
>>> S
'AÄBèC'

>>> S = 'A\u00C4B\u000000E8C'        # 4- and 8-digit Unicode escapes (str only)
>>> S
'AÄBèC'

>>> B = b'A\xc4B\xe8C'                # bytes recognizes hex escapes, but not Unicode
>>> B
b'A\xc4B\xe8C'

>>> B = b'A\u00C4B\u000000E8C'        # Unicode escape sequences taken literally!
<stdin>:1: SyntaxWarning: invalid escape sequence '\u'
>>> B                                  # bytes is encoded bytes, not code points
b'A\\u00C4B\\u000000E8C'

>>> B = b'A\xc4B\xe8C'                # Use hex escapes for latin-1 bytes
>>> B                                  # Prints non-ASCII bytes as hex
b'A\xc4B\xe8C'

```

```
>>> print(B)                                # For both interactive and print()
b'A\xc4B\xe8C'
>>> B.decode('latin-1')                    # Decode to str to interpret as text
'AÄBèC'
```

Finally, notice that bytes literals assume that textual characters embedded within them are ASCII and require escapes for byte values > 127. By contrast, str literals in code like that in the following allow embedding any character supported by the source code encoding of the hosting file or GUI (as you'll learn in a moment, the encoding used for code defaults to UTF-8, sans declarations in the code's file):

```
>>> S = 'AÄBèC'                            # Chars from UTF-8 if no encoding declaration
>>> S                                        # Decoded to str when code is read by Py
'AÄBèC'

>>> B = b'AÄBèC'
SyntaxError: bytes can only contain ASCII literal characters.

>>> B = b'A\xc4B\xe8C'                     # Chars must be ASCII, or hex escapes
>>> B                                        # Non-ASCII's are latin-1 encoded bytes
b'A\xc4B\xe8C'
>>> B.decode('latin-1')
'AÄBèC'

>>> S.encode()                            # Source code encoded per UTF-8 by default
b'A\xc3\x84B\xc3\xa8C'                    # Methods use UTF-8 to encode, unless passed
>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'
>>> B.decode()                            # Raw bytes do not correspond to utf-8
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc4 in position 1:...

>>> S = 'AÄBèC'
>>> S.encode()                            # Method's default utf-8 encoding
b'A\xc3\x84B\xc3\xa8C'
>>>
>>> T = S.encode('cp500')                  # "Convert" to EBCDIC bytes
>>> T
b'\xc1c\xc2T\xc3'
>>>
>>> U = T.decode('cp500')                  # Back to Unicode code points
>>> U
'AÄBèC'
>>>
>>> U.encode()                            # Back to UTF-8 bytes, by default
b'A\xc3\x84B\xc3\xa8C'
```

Notice how the last part of the preceding code seems to “convert” encodings from UTF-8 to cp500 and back again. Really, this just creates different encoded representations of the same Unicode code points, but this pattern can be used to translate encoded text when needed. Text in a file, for instance, can be re-encoded with a decode (to str) plus an encode (to bytes) combination that changes its stored encoding; as you'll see ahead, the open function does most of this work for you.

Also, note how the preceding code is able to use a str literal 'AÄBèC' with raw Unicode characters for its non-ASCII characters. This is noticeably simpler than coding escapes and works as long as your code file (and GUI) support it, as the next section will explain.

Source-File Encoding Declarations

Unicode escapes suffice for the occasional Unicode character in string literals, but they can become tedious if you need to code non-ASCII text in your strings frequently. For string literals and other text that you embed in your script files (or paste into your coding GUI), Python uses the *UTF-8* encoding by default to read your code's text but allows you to change this per file to use an arbitrary encoding. With this support, your code can directly embed any unescaped characters that the chosen encoding supports.

To make this work, simply use Python's default UTF-8 encoding to save your source code file in your text editor, or include a comment that names the Unicode encoding that you used for the save if it differs. This special encoding-declaration comment must appear as either the first or second line in your script (e.g., a `#!` line works before it: see [Appendix A](#)) and is usually of the following form (see Python's manuals for other forms it accepts):

```
# -*- coding: latin-1 -*-
```

When present, Python will recognize text in your code represented natively (unescaped) in the given encoding. That way, you can edit your script file in a text editor that accepts, displays, and saves accented and other non-ASCII characters, and Python will correctly decode them when reading your string literals and other program-file text.

For example, notice the coding comment at the top of [Example 37-1](#): when this file is saved in the non-default Latin-1 encoding, it allows Python to recognize Latin-1 characters embedded in the string literal to be assigned to `myStr1` in the text of the source file. This file also neatly summarizes the various ways to code non-ASCII text in Python.

Example 37-1. source-encoding-latin1.py

```
# -*- coding: Latin-1 -*-

#-----
# Demo all the ways to code non-ASCII text in Python, plus source encodings.
#
# If this file is saved as Latin-1 text, it works as is. But changing the
# coding line above to either ASCII or UTF-8 will then fail because the
# Latin-1 0xc4 and 0xe8 saved in myStr1's value are not valid in either.
#
# A UTF-8 line works if this file is also saved as UTF-8 to make its myStr1
# text match. Because UTF-8 is the default for source, the line above is
# optional if the file is saved as UTF-8 or its text is all UTF-8 compatible
# (e.g., ASCII, which is a subset of both the Latin-1 and UTF-8 encodings).
#-----

myStr1 = 'AÃBèC'                                # Raw, per source encoding

myStr2 = 'A\xc4B\xe8C'                          # Hex code-point escapes

myStr3 = 'A\u00c4B\u0000000e8C'                 # Unicode short/long escapes

myStr4 = 'A' + chr(0xc4) + 'B' + chr(0xe8) + 'C' # Concatenated code points

import sys, locale
print('Sys hosting platform: ', sys.platform)
print('Sys default encoding: ', sys.getdefaultencoding())
print('Open default encoding: ', locale.getpreferredencoding(False))
```

```

for aStr in (myStr1, myStr2, myStr3, myStr4):
    print(f'{aStr}, strlen={len(aStr)}', end=', ')    # Decoded text+length

    bytes1 = aStr.encode()                # Default UTF-8: 2 bytes for accents
    bytes2 = aStr.encode('latin-1')       # Explicit Latin-1: 1 byte per char
    #bytes3 = aStr.encode('ascii')         # ASCII fails: outside 0...127 range

    print(f'byteslen1={len(bytes1)}, byteslen2={len(bytes2)}')    # Encoded length

```

After saving this file in a text editor with encoding Latin-1 (or its default cp1252 superset on some Windows), running it as a script prints its four strings, their character code-point lengths, and their byte lengths in two encodings that work—the encode method’s default UTF-8, and an explicit Latin-1 (ASCII is too narrow to use). The Python default encodings it also prints may vary across host platforms, but the rest of the output will not:

```

$ python3 source-encoding-latin1.py
Sys hosting platform: darwin
Sys default encoding: utf-8
Open default encoding: UTF-8
AÃBèC, strlen=5, byteslen1=7, byteslen2=5
AÃBèC, strlen=5, byteslen1=7, byteslen2=5
AÃBèC, strlen=5, byteslen1=7, byteslen2=5
AÃBèC, strlen=5, byteslen1=7, byteslen2=5

```

To change this file to use UTF-8 instead, first save it with the UTF-8 encoding in your text editor or by running Python code like the following to make its embedded literal match (you’ll learn how and why this code works when we explore Unicode text files ahead):

```

>>> text = open('source-encoding-latin1.py', encoding='latin1').read()
>>> open('source-encoding-utf8.py', 'w', encoding='utf8').write(text)

```

Then, either mod its first line to name UTF-8, or delete the first line altogether (UTF-8 is Python’s default for source code). After you’re done, the only difference in the two versions of the source file will be the way the embedded Unicode literal is rendered on your platform; here’s the verdict on the UTF-8-centric macOS using its diff to compare (use fc instead on Windows):

```

$ diff source-encoding-latin1.py source-encoding-utf8.py
1c1
< # -*- coding: Latin-1 -*-
---
> # -*- coding: UTF-8 -*-
13c13
< myStr1 = 'A?B?C'
---
> myStr1 = 'AÃBèC'

$ python3 source-encoding-utf8.py    # Same output as Latin-1 version above

```

Since most programmers are likely to fall back on the default and general (really, *universal*) UTF-8 encoding in Python, we’ll defer to Python’s standard [manual set](#) for more details on this option, as well as its more advanced and obscure Unicode support such as properties and character-name escapes in strings that we’ll skip here.



Unicode in variable names: Source-file encoding declarations apply to a file's content in general and support arbitrary kinds of text. The rules for variable names within a file's code, however, are more stringent.

As noted briefly in [Chapter 11](#), Python allows some, but not all, non-ASCII Unicode characters to be used for variables in your code. Roughly, number- and letter-like characters work, but symbols and emojis are not allowed. For instance, `häck` is a valid variable, but `häck🐍` is not. You can check whether a specific string passes as a variable with the `isidentifier` method of `str`, but the rules behind this are complex and best had in Python's language manual.

Also, keep in mind that, even when valid, non-ASCII in variables may make your code difficult to use on some keyboards and devices outside a given language's locale. In fact, all code in the Python standard library must use ASCII-only identifiers for this reason. As usual, use with care.

Using Byte Strings

We'll be able to see strings in action again when we study files ahead. First, though, let's take a brief side trip to dig a bit deeper into the operation sets provided by objects geared for binary data—the `bytes` type and its `bytearray` mutable kin. While these types can be used to hold encoded text too (as in prior sections), their scope is much broader: anything that can be stored as bytes works, and that's everything digital.

As mentioned earlier, Python's `bytes` type supports sequence operations and most of the same methods available on `str`. Even so, because you cannot mix and match bytes and `str` without explicit conversions, you'll generally use `str` objects and text files for text data and `bytes` objects and binary files for binary data. This makes bytes a crucial tool in many roles and worthy of a quick demo here.

Methods

If you really want to see what attributes `str` has that `bytes` doesn't, you can always check their `dir` results (review: `set(X) - set(Y)` is items in `X` but not in `Y`). This can also tell you something about the expression operators they support (e.g., `__mod__` and `__rmod__` implement the `%` operator, and they're present in both today):

```
$ python3
Python 3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024,...

# Attributes unique to str

>>> sorted(set(dir('abc')) - set(dir(b'abc')))
['casefold', 'encode', 'format', 'format_map', 'isdecimal', 'isidentifier',
'isnumeric', 'isprintable']

# Attributes unique to bytes

>>> sorted(set(dir(b'abc')) - set(dir('abc')))
['__buffer__', '__bytes__', 'decode', 'fromhex', 'hex']
```

As you can see, `str` and `bytes` have almost identical functionality; their unique attributes are generally methods that don't apply to the other (format is an outlier you'll meet shortly). For instance, `decode` translates a raw bytes into its `str` representation, and `encode` translates a `str` into its raw bytes

representation. Most other methods are shared between the two types. Moreover, bytes are immutable just like `str`:

```
>>> B = b'code'           # b'...' bytes literal
>>> B.find(b'od')          # Search for substr offset
1

>>> B.replace(b'od', b'XY') # New bytes with replacement
b'cXYe'
>>> B
b'code'

>>> B[0] = 'x'
TypeError: 'bytes' object does not support item assignment
```

For more bytes methods, see the earlier coverage of string fundamentals in [Chapter 7](#); bytes do most of the same work, though their methods generally return a new bytes instead of a `str`.

Sequence Operations

Besides method calls, all the usual generic sequence operations you know from other sequences, like lists, work as expected on both `str` and `bytes`. This includes indexing, slicing, concatenation, and so on. As we've learned, `str` is a sequence of character code points, and `bytes` is a sequence of byte-size integers, but their sequence operations' semantics are the same.

Notice in the following, though, that indexing bytes returns an integer giving the byte's binary value; `bytes` really is a sequence of 8-bit integers in the 0...255 range, but when displayed, its components print as either ASCII characters if their values fall into ASCII's 0...127 code-point range, or as hex escapes otherwise:

```
>>> B = b'code'           # bytes and str are both sequences
>>> B                     # Bytes in the 0...127 range display as ASCII
b'code'

>>> B[0], B[-1]           # Indexing: returns a byte's integer value
(99, 101)
>>> 'code'[0]             # But indexing a str returns a 1-item str
'c'

>>> chr(B[0])             # Unicode character (code point) for byte's value
'c'
>>> list(B)               # But it's really integer bytes
[99, 111, 100, 101]

>>> b'A\x42C\xff\x63'     # That happen to display as ASCII if in 0...127
b'ABC\xffc'
>>> chr(0x63), hex(B[0])  # And accept hex escapes for byte values
('c', '0x63')

>>> B[1:], B[:-1]         # Slicing: bytes (that display 0...127 as ASCII)
(b'ode', b'cod')
>>> len(B)                # Length: number bytes (not necessarily characters)
4

>>> B + b'lmn'           # Concatenation: bytes
b'code\u006c\u006d\u006e'
>>> B * 4                # Repetition: bytes
b'codecodecodecode'
```

Formatting

One notable exception to the same-operations rule for strings: string formatting % expressions work on bytes too (as of Python 3.5), but neither the `format` method nor `f'...'` f-string formatting is available for bytes—an odd bifurcation that seems to forget that formatting comes in multiple flavors today:

```
>>> b'a %s string' % b'fine'                                # Py 3.5+ formatting for bytes
b'a fine string'
>>> b'a %s string' % bytes([0xFF, 0xFE])                    # Non-ASCII bytes work too
b'a \xff\xfe string'

>>> 'a {}'.format('fine')                                    # But format method only for str
'a fine string'
>>> b'a {}'.format(b'fine')
AttributeError: 'bytes' object has no attribute 'format'

>>> kind = 'fine'
>>> f'a {kind} string'                                       # But f-strings only for str
'a fine string'
>>> bf'a {kind} string'
SyntaxError: invalid syntax
```

There are arguably sound reasons that formatting shouldn't work on bytes—it's just raw bytes, after all, which happens to accept and print ASCII characters as their ASCII code-point values, and may use any encoding if it's text, or none at all if it's not. Plugging ASCII text into an encoded UTF-16 string or loaded image, for instance, makes no sense. But these sound reasons are inconsistently violated for the sake of the % expression alone.

Moreover, post-operation conversion by encoding isn't the same as bytes operations, and will fail if the default or explicit encoding isn't inclusive enough to handle the text:

```
>>> kind = 'fine'
>>> f'a {kind} string'.encode()                               # Encoding != bytes operations
b'a fine string'

>>> kind = 'AÄBèC'
>>> f'a {kind} string'.encode('ascii')                       # And narrow encodings may fail
UnicodeEncodeError: 'ascii' codec can't encode character '\xc4' in position 3:...
```

This inconsistency is prone to change over time, but today, the embarrassment of riches in the formatting department has also given birth to an embarrassment of special cases.

Other Ways to Make Bytes

So far in this section, we've been making bytes objects with the `b'...'` literal syntax, but they can also be created by calling the bytes constructor with a `str` and an encoding name, by calling bytes with an iterable of integers representing byte values, or by encoding a `str` object per the default (or passed-in) encoding. We met some of these earlier in the guise of conversions, but they're more general than previously told.

For example, encoding takes a `str` and returns the raw binary bytes value of the string according to its encoding specification; decoding takes a raw bytes sequence and encodes it to its string representation—a series of Unicode characters:

```
>>> B = b'abc'
>>> B
b'abc'
```

```

>>> B = bytes('abc', 'ascii')
>>> B
b'abc'

>>> ord('a')
97
>>> B = bytes([97, 98, 99])
>>> B
b'abc'

>>> B = 'code'.encode()      # Or bytes()
>>> B
b'code'

>>> S = B.decode()           # Or str()
>>> S
'code'

```

As we saw earlier, the last two of these operations can also be thought of as tools for converting between str and bytes, as expanded upon in the next section.

Mixing String Types

When we used the `replace` earlier when sampling bytes methods, you may have noticed that we had to pass in two bytes objects for `from` and `to`—str types won't work there. More generally, Python requires specific string types in some contexts and expects manual conversions if needed. Here's the story for function and method calls:

```

>>> B = b'code'

>>> B.replace('od', 'XY')
TypeError: a bytes-like object is required, not 'str'

>>> B.replace(b'od', b'XY')
b'cXYe'

>>> B.replace(bytes('od'), bytes('XY'))
TypeError: string argument without an encoding

>>> B.replace(bytes('od', 'ascii'), bytes('XY', 'utf-8'))
b'cXYe'

```

The same holds for mixed-type expressions: you should try to keep your text and binary data separate, but if you must mix, you generally also must convert:

```

>>> b'ab' + 'cd'
TypeError: can't concat str to bytes

>>> b'ab'.decode() + 'cd'      # bytes to str
'abcd'

>>> b'ab' + 'cd'.encode()      # str to bytes
b'abcd'

>>> b'ab' + bytes('cd', 'ascii') # str to bytes
b'abcd'

```

Two notes here. First, remember that encoding and decoding are more than a simple type conversion; as we learned in the coverage earlier, they create different types of data altogether. Second, although you can create bytes objects yourself to represent packed binary data, they can also be made automatically by reading files opened in binary mode, as we will later in this chapter. First, though, let's briefly explore bytes' elusive and changeable colleague.

The bytearray Object

So far in this chapter, we've focused on `str` and `bytes` because they will be your go-to string tools. Python, however, has a third string type—`bytearray`, which is essentially a mutable variant of `bytes`, and thus a mutable sequence of integers in the range 0...255. As such, it supports the same string methods and sequence operations as `bytes`, as well as the mutable in-place-change operations found on lists.

We've already seen most operations that apply to `bytearray`, so we'll just take a quick tour here to sample their flavor. First off, you can call `bytearray` as a function passing a bytes (not a `str`) to make a new mutable sequence of small (0...255) integers:

```
>>> B = b'code'                # A str 'code' does not work: not bytes
>>> C = bytearray(B)
>>> C
bytearray(b'code')
>>> C[0], chr(C[0])              # ASCII code-point integer for 'c'
(99, 'c')
```

Once you've got a `bytearray`, you can change it in place using the same sorts of operations available to modify a list, but keep in mind that you must assign just integers to its cells, not `str` text strings, and not arbitrary objects:

```
>>> C[0] = 'x'
TypeError: 'str' object cannot be interpreted as an integer

>>> C[0] = b'x'
TypeError: 'bytes' object cannot be interpreted as an integer

>>> C[0] = ord('x')
>>> C
bytearray(b'xode')

>>> C[1] = b'Y'[0]
>>> C
bytearray(b'xYde')
```

The `bytearray`'s methods set overlaps broadly with both `str` and `bytes` because it's a kind of string sequence, but it also has the list object's in-place change methods because it's mutable too (the second of the following means attributes unique to `bytearray`):

```
>>> sorted(set(dir(b'abc')) - set(dir(bytearray(b'abc'))))
['__bytes__', '__getnewargs__']

>>> sorted(set(dir(bytearray(b'abc')) - set(dir(b'abc'))))
['_alloc_', '_delitem_', '_iadd_', '_imul_', '_release_buffer_',
'_setitem_', 'append', 'clear', 'copy', 'extend', 'insert', 'pop', 'remove', 'reverse']
```

Hence, it's something of a combo platter—you get methods for in-place changes:

```
>>> C
bytearray(b'xYde')
```

```

>>> C.append(b'LMN')
TypeError: 'bytes' object cannot be interpreted as an integer

>>> C.append(b'LMN'[0])
>>> C
bytearray(b'xYdeL')

>>> C.append(ord('M'))
>>> C
bytearray(b'xYdeLM')

>>> C.extend(b'N0')
>>> C
bytearray(b'xYdeLMN0')

```

Plus all the usual sequence operations and string methods:

```

>>> C + b'!#'
bytearray(b'xYdeLMN0!#')

>>> C[0], chr(C[0])
(120, 'x')

>>> C[1:]
bytearray(b'YdeLMN0')

>>> len(C)
8

>>> C
bytearray(b'xYdeLMN0')

>>> C.replace('xY', 'co')
TypeError: a bytes-like object is required, not 'str'

>>> C.replace(b'xY', b'co')
bytearray(b'codeLMN0')

>>> C
bytearray(b'xYamLMN0')

>>> C * 4
bytearray(b'xYdeLMNOxYdeLMNOxYdeLMNOxYdeLMNO')

```

For completeness, this section would be remiss if it didn't call out that you can also make a `bytes` or `bytearray` by passing in any sort of integer sequence or generator, not just text strings. This also provides a sort of conversion from an integer character code to a `bytes`, but only if the integer is in the range 0...255, and only if it's embedded in a sequence (else you get a `bytes` with that many zeroes—surprisingly!). See Python's manuals for more esoteric bits that we don't have space to cover here:

```

>>> bytes([1, 2, 3, 4]), bytearray([1, 2, 3, 4])
(b'\x01\x02\x03\x04', bytearray(b'\x01\x02\x03\x04'))

>>> bytes([115])           # int => bytes
b's'

>>> bytes([115])[0]        # bytes => int
115

>>> bytes([999])           # Too big for a byte
ValueError: bytes must be in range(0, 256)

```

```
>>> bytes(115)           # Lots of zeroes!
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00...'

>>> bytes(range(97, 104)), bytes(c + 97 for c in range(7))    # Generators too
(b'abcdefg', b'abcdefg')
```

Finally, by way of summary, the following examples demonstrate how bytes and bytearray are sequences of integers, and str is a sequence of characters (i.e., decoded Unicode code points); although all three can contain textual content and support many of the same operations, you should use str for decoded text, bytes for binary data including encoded text, and bytearray for binary data you wish to change in place to avoid the time and space overheads of generating copies for each change you make:

```
>>> B = b'code' # Bytes
>>> list(B)
[99, 111, 100, 101]

>>> C = bytearray(b'code') # Changeable bytes
>>> list(C)
[99, 111, 100, 101]

>>> S = 'code' # Unicode text
>>> list(S)
['c', 'o', 'd', 'e']
```

Using Text and Binary Files

Now that we’ve learned all about Python’s string types, let’s return to their roles in files—the main context in which most programmers will likely encounter Unicode and bytes.

As mentioned earlier, the *mode* in which you open a file is crucial in Python: it determines both how the file's content is interpreted as well as the object type you will use to process that content in your script. By way of review, text mode implies `str` objects and binary mode implies `bytes`, as follows:

Text-mode files

Interpret file contents according to an encoding—either the default for your platform or one whose name you pass in to `open`. By passing in an encoding name, you can force conversions for various types of Unicode files. Text-mode files may also handle BOM headers for some encodings (deferred till the end of this chapter) and may perform universal newline translations for you or not; by default, all newline forms map to the `\n` character in your script, regardless of which platform you are on.

Binary-mode files

Instead return file content to you raw as a sequence of integers representing byte values, with no encoding or decoding, no BOM handling, and no newline translations.

In terms of code, the second positional argument to `open` (a.k.a. `mode` when passed by keyword) determines whether you want text or binary processing and types—adding a `b` to the mode string implies binary mode. The default mode is `rt`, which is the same as `r`, and means text input. In addition, the mode argument to `open` also implies an object type for file content representation regardless of the underlying platform—text files return a `str` for reads and expect one for writes, but binary files return a `bytes` for reads and expect `bytes` (or `bytearray`) for writes.

Text-File Basics

To demonstrate, let's review basic file I/O. As long as you're processing simple text files that adhere to your platform's default encoding, files look and feel much as they do in this book's earlier coverage (for that matter, so do strings in general). The next example, for instance, writes one line of text to a file and reads it back:

```
>>> file = open('temp.txt', 'w')      # Use default encoding on host, mode w=write
>>> size = file.write('abc\n')        # Returns number characters written
>>> file.close()                      # Manual close to flush output buffer

>>> file = open('temp.txt')            # Default mode is "r" == "rt": text input
>>> text = file.read()
>>> text
'abc\n'
```

As a refresher, the first argument to `open` is the file's *pathname*—the address of a file in the host's folder hierarchy that's either absolute or relative to the current directory. The second argument to `open` is *mode*—where `w` means write text, and the default means read it, and `write` methods return the number of written *items*—either characters for text mode or bytes for binary mode. Also, the `close` call here is optional in some contexts (e.g., the widely used *CPython* auto-closes files when their objects are garbage collected) but is generally advised to flush changes and avoid memory growth.¹

Technically, the preceding example writes and reads Unicode text, but it's hardly noticeable: the ASCII text string is encoded and decoded per the hosting platform's encoding default. We're also relying on platform-agnostic newline handling for `\n`, but we must move ahead for more on encodings and newlines.

Text and Binary Modes

Next, let's write a *text file* and read it back in both text and binary modes. Notice in the following how text mode requires us to provide a `str` for writing, `rb` distinguishes binary-mode input, and reading gives us a `str` or bytes depending on the mode (opens and transfer operations are strung together here into one-liners just for brevity; again, remember to `close` explicitly in production code, and possibly in some IDEs and outside *CPython*):

```
$ py -3                                # Run on Windows (no |r on Unix)
>>> open('temp.txt', 'w').write('abc\n') # Text-mode output, provide a str
4

>>> open('temp.txt', 'r').read()         # Text-mode input, returns a str
'abc\n'

>>> open('temp.txt', 'rb').read()        # Binary-mode input, returns a bytes
b'abc\r\n'
```

1 File fine points: most of this chapter's file examples use filenames relative to, and hence stored in, the current directory, so be sure to run them in a directory (a.k.a. folder) where you have *permission* to create files; `cd` to one in your shell or IDE if needed. This may matter on platforms with major storage constraints like Android, but you'll probably already be in a safe folder in most apps. Also, file *extensions* (e.g., `.txt`) don't mean anything to Python and are technically optional; some IDEs that hang on to objects for debugging may require manual `close` calls to flush changes too; and purists take note that `file` is no longer a built-in name in Python and OK to use as a variable here!

Observe how the *newline character* is always `\n` when writing and reading in text mode with `str` but `\r\n` when reading in binary mode with `bytes`. This reflects the fact that this was run on Windows. Though it has nothing to do with Unicode, text-mode files automatically map all `\n` in `str` to and from the host platform's newline separator: `\r\n` on Windows and just `\n` on Unix. When reading in binary mode, though, we get what's actually in the file—with neither newline mapping nor Unicode decoding.

Now, let's do the same, but with a *binary file*. We provide a `bytes` to write and still get back a `str` or `bytes` depending on the input mode, though the `\n` isn't expanded to `\r\n` on Windows this time:

```
>>> open('temp.bin', 'wb').write(b'abc\n')    # Binary-mode output, send a bytes
4

>>> open('temp.bin', 'r').read()              # Text-mode input, receive a str
'abc\n'

>>> open('temp.bin', 'rb').read()             # Binary-mode input, bytes sans mapping
b'abc\n'
```

This holds true even if the data we're writing to the binary file is truly binary in nature. In the following, the `\x00` is a binary zero byte and not a printable character, though it works in the middle of a `bytes` and qualifies as a text code point in the default encoding (strictly speaking, zero is a character called null or NUL in ASCII and its supersets like UTF-8):

```
>>> open('temp.bin', 'wb').write(b'a\x00c')    # Binary data included
3

>>> open('temp.bin', 'r').read()              # Text-mode: str with NUL
'a\x00c'

>>> open('temp.bin', 'rb').read()             # Binary mode: bytes
b'a\x00c'
```

Binary mode files always return contents as a `bytes` object but accept either a `bytes` or `bytearray` object for writing. This naturally follows, given that `bytearray` is mostly just a mutable variant of `bytes`. In fact, most APIs in Python that accept a `bytes` also allow a `bytearray` (the `bytes` here are also ASCII characters too obscure for this note):

```
>>> BA = bytearray(b'\x01\x02\x03')
>>> open('temp.bin', 'wb').write(BA)
3

>>> open('temp.bin', 'r').read()
'\x01\x02\x03'

>>> open('temp.bin', 'rb').read()
b'\x01\x02\x03'
```

Finally, notice that you can't get away with violating Python's `str/bytes` (i.e., text/binary) distinction when it comes to files; in the following, we get errors if we try to write a `bytes` to a text file or a `str` to a binary file. Remember, although it is often possible to convert between these two types (as described earlier in this chapter), you will usually want to stick to `str` for text data and `bytes` for binary data:

```
>>> open('temp.txt', 'w').write('abc\n')      # Auto encodes str to bytes
4

>>> open('temp.txt', 'w').write(b'abc\n')     # But bytes != decoded text
TypeError: write() argument must be str, not bytes

>>> open('temp.bin', 'wb').write(b'abc\n')    # Writes raw bytes
4
```



```
>>> open('temp.bin', 'wb').write('abc\n')           # But str != raw bytes
TypeError: a bytes-like object is required, not 'str'
```

This may seem strict, but Python cannot guess how you wish to interpret the contents of a `bytes` or `str` when used in the opposite context and wisely refuses to convert implicitly (a `bytes` might be an image, after all). Moreover, because `str` and `bytes` operation sets largely intersect, the choice of types won't be much of a dilemma for most programs. Watch for the `struct` module coverage ahead for another binary-file example.

Unicode-Text Files

And now for the featured attraction of our files tour: text files with non-ASCII text. Beyond their text/binary distinction, Python files come with additional requirements and tools for dealing with Unicode text. In short, text files allow a specific Unicode encoding-scheme name to be passed in with an `encoding` argument to `open` and use it to automatically *decode* and *encode* text on input and output, respectively. As abstract examples, for a file identified by a *pathname* string:

```
open(pathname, 'r', encoding='utf8')
    Returns a file object that decodes text from UTF-8 on reads
```

```
open(pathname, 'w', encoding='latin1')
    Returns a file object that encodes text to Latin-1 on writes
```

The file object returned by the first of the preceding assumes the file's content is encoded per UTF-8 and automatically decodes it to `str` Unicode code points when read by the program. Similarly, the result of the second line of code encodes `str` code points to their Latin-1 format as they are output to the file. Here's the text-file story with the universal *UTF-8* encoding and three non-ASCII characters in the content:

```
>>> file = open('uni.txt', 'w', encoding='utf8')      # Auto encodes to bytes
>>> file.write('♥ 2 hÅck 🐍')
10
>>> file.close()

>>> text = open('uni.txt', 'r', encoding='utf8').read() # Auto decodes to str
>>> text
'♥ 2 hÅck 🐍'

>>> [ord(c) for c in text]                             # Character code points
[128155, 32, 50, 32, 104, 196, 99, 107, 32, 128013]

>>> raw = open('uni.txt', 'rb').read()                 # No decoding applied
>>> raw
b'\xf0\x9f\x92\x9b 2 h\xc3\x84ck \xf0\x9f\x90\x8d'
```

File transfers raise exceptions whenever a requested encoding doesn't work, so the encoding you pass must match the data. For example, *ASCII* is not inclusive enough to handle the augmented and emoji characters in the text we're writing here and fails on both reads and writes:

```
>>> open('uni.txt', 'r', encoding='ascii').read()
UnicodeDecodeError: 'ascii' codec can't decode byte 0xf0 in position 0:...

>>> open('ascii.txt', 'w', encoding='ascii').write('♥ 2 hÅck 🐍')
UnicodeEncodeError: 'ascii' codec can't encode character '\U0001f49b'...

>>> hex(ord('♥'))   # ASCII will always break your heart?
'0x1f49b'
```

By contrast, using the broader and more general *UTF-16* on both ends handles this text in full, though its encoded bytes stored in the file naturally differ from those of *UTF-8*:

```
>>> file = open('uni2.txt', 'w', encoding='utf16')          # UTF-16 works too
>>> file.write('♥ 2 häck 🐸')
10
>>> file.close()

>>> text = open('uni2.txt', 'r', encoding='utf16').read()    # Files encode+decode
>>> text
'♥ 2 häck 🐸'

>>> [ord(c) for c in text]                                    # Same code points
[128155, 32, 50, 32, 104, 196, 99, 107, 32, 128013]

>>> open('uni2.txt', 'rb').read()                             # Different encoding
b'\xff\xfe\xdb\x9b\xdc \x02\x00 \x0h\x00\x04\x0c\x00k\x00 \x00=\xdb\r\xdc'
```

Even for broader encodings like *UTF-8* and *UTF-16*, though, you cannot *mix and match*: using an incompatible encoding still won't work because encoded bytes in the file differ. Although you can sometimes handle unknown encodings with binary-mode files, open error handlers, and other techniques, your encoding must generally match your file's data:

```
>>> open('uni.txt', 'r', encoding='utf16').read()
UnicodeDecodeError: 'utf-16-le' codec can't decode byte 0x8d in position 16:...

>>> open('uni2.txt', 'r', encoding='utf8').read()
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 0:...
```

In the *absence* of an encoding argument, text files still encode and decode per a host- and platform-specific default in locale discussed earlier. But as a reminder: although you may not notice these translations if your default and files agree, you generally should not rely on the default; it makes your programs dependent on the context in which their files were created and can lead to portability issues (and nightmares; see the upcoming sidebar “Unicode Defaults and *UTF-8 Mode*” on page 982).

For instance, a program run on a *UTF-8* default platform (the macOS and Android norm) may have trouble using a file made under a *cp1252* default (the ASCII-superset default on some Windows hosts) and vice versa, and content fetched from other devices may be encoded arbitrarily. Use explicit open encodings as a rule. This may also help if environment settings are not reliable (e.g., when running as a generic user for security in server-side web scripts).

All that being said, you'll probably find files to be easier in practice than the full details may suggest. Accessing web pages and images, for example, soon becomes second nature and simple. For variety, the following first omits mode (again, its default is the same as *r*) and then passes mode by explicit keyword (instead of position), and this example is abstract (substitute pathnames of real and accessible files on your device to run live):

```
>>> text = open('Websites/about-lp5e.html', encoding='utf8').read()
>>> text[:79]
'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.o'

>>> image = open('Websites/lp5e-large.jpg', mode='rb').read()
>>> image[:20]
b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x00\x00\x01\x00\x01\x00\x00'
```

Once you've loaded content this way, all the *str* and *bytes* operations we've seen in this chapter are at your disposal for wrangling text and binary data, and saving the result follows the same pattern—simply use an encoding for text and binary mode for bytes:

```
>>> text = text.replace('2013', '2024')
>>> open('Websites/about-lp6e.html', mode='w', encoding='utf8').write(text)

>>> image = some_sort_of_modding(image)
>>> open('Websites/lp6e-large.jpg', mode='wb').write(image)
```

You might mod images, for example, with the many tools provided by the third-party *Pillow* (f.k.a. PIL) imaging library for Python or similar. With Python's files and strings, content possibilities are largely endless.

In the interest of full disclosure, Python's `open` accepts additional arguments that modify its behavior. Among them, `newline` changes newline mapping; `errors` specifies handling of encoding and decoding errors (e.g., 'surrogateescape' replaces failing bytes with sequences that can be used to restore them on output); and `buffering` alters, well, buffering. Because their defaults are generally what you'll use, we'll defer to the Python standard-library manual for the fine print on these and other advanced file options.



Blast from the past: Python's `codecs` module also has an `open` that returns an auto-decode/encode file object just like the built-in `open` function. This was used for Python 2.X backward compatibility in times gone by, but there's no obvious reason to rely on it in new code. Still, you might see it in legacy code (and code that relies on any of its unique behaviors) anyhow:

```
>>> import codecs
>>> f = codecs.open('uni.txt', 'r', encoding='utf8')
>>> f.read()
'♥ 2 hÄck 🐍'
```

Unicode, Bytes, and Other String Tools

Besides built-in strings and files, many of the popular string-processing tools in Python's standard library installed with Python itself also adhere to the `str`/bytes dichotomy. We won't cover any of these application-focused topics in detail in this core-language book, but as a sample, here's a *very* brief look at how some of these tools handle the split. See Python's Library Reference for more on the tools used here if any pique your interest.

The `re` Pattern-Matching Module

Python's `re` pattern-matching module has been generalized to work on objects of any string type—`str`, `bytes`, and `bytearray` (a `(.*)` means any run, saved as a group):

```
>>> import re
>>> S = '🐍 is the fastest way to 🍕!'
>>> B = b'Python is the fastest way to pizza!'

>>> re.match('(.*) the (.*) way (.*)', S).groups()      # str + str => str
('🐍 is', 'fastest', 'to 🍕!')

>>> re.match(b'(.*) the (.*) way (.*)', B).groups()    # bytes + bytes => bytes
(b'Python is', b'fastest', b'to pizza!')

>>> re.match(b'(.*) the (.*) way (.*)', bytearray(B)).groups()
(b'Python is', b'fastest', b'to pizza!')
```

Like many tools, though, its result types depend on the type of strings you pass in, and you can't mix `str` and `bytes` types in its calls' arguments (sans explicit conversions, of course); `bytearray` is also unusable here as a pattern because it's mutable (and changeable means “unhashable”):

```
>>> re.match('(.*) the (.*) way (.*)', B).groups()
TypeError: cannot use a string pattern on a bytes-like object
>>> re.match(b'(.*) the (.*) way (.*)', S).groups()
TypeError: cannot use a bytes pattern on a string-like object

>>> re.match('(.*) the (.*) way (.*)', bytearray(B)).groups()
TypeError: cannot use a string pattern on a bytes-like object
>>> re.match(bytearray(b'(.*) the (.*) way (.*)'), bytearray(B)).groups()
TypeError: unhashable type: 'bytearray'
```

The struct Binary-Data Module

The Python `struct` module, used to create and extract packed binary data from strings, operates on `bytes` and `bytearray` only, not `str`—which makes sense, given that it's intended for processing binary data, not decoded text. This module uses a format string to specify the types and sizes of objects to pack to and from a bytes string, along with an endianness (i.e., significant side of bitstrings) specifier like `>` for big-endian:

```
>>> import struct
>>> B = struct.pack('>i4sh', 7, b'code', 8)    # int , bytes(4s), int(h) => bytes
>>> B
b'\x00\x00\x00\x07code\x00\x08'

>>> vals = struct.unpack('>i4sh', B)           # Packed data is bytes, not str
>>> vals
(7, b'code', 8)

>>> vals = struct.unpack('>i4sh', B.decode())
TypeError: a bytes-like object is required, not 'str'
```

You'll often use this in conjunction with binary-mode files to make the packed bytes persistent across program runs (e.g., for saving and restoring app user settings):

```
>>> F = open('data.bin', 'wb')                # Open binary output file
>>> data = struct.pack('>i4sh', 7, b'code', 8) # Create packed binary data
>>> data                                       # bytes, not str
b'\x00\x00\x00\x07code\x00\x08'

>>> F.write(data)                             # Write to the file
10

>>> F.close()                                # Flush changes

>>> F = open('data.bin', 'rb')                # Open binary input file
>>> data = F.read()                          # Read bytes
>>> data
b'\x00\x00\x00\x07code\x00\x08'

>>> values = struct.unpack('>i4sh', data)      # Extract packed binary data
>>> values                                   # Back to Python objects
(7, b'code', 8)
```

The pickle and json Serialization Modules

Python's `pickle` module provides another way to save data in files but allows saved data to be nearly arbitrary Python objects instead of values coerced to bytes as in `struct`. We met this module briefly in Chapters 9, 28, and 31. For completeness here, keep in mind that the `pickle` module always creates a

bytes object, regardless of the default or passed-in protocol (data format level). You can see this by using the module's `dumps` call to return an object's pickle string:

```
>>> import pickle                                # dumps() returns pickle string
>>> pickle.dumps(['code', 4, '🐍'])              # Default protocol=binary
b'\x80\x04\x95\x15\x00\x00\x00\x00 ...etc... \x04\x8c\x04\xf0\x9f\x90\x8d\x94e.'
```

```
>>> pickle.dumps(['code', 4, '🐍'], protocol=0)    # ASCII protocol 0, still bytes!
b'(\x00\nVcode\np1\naI4\naV\\U0001f40d\np2\na.'
```

This implies that files used to store pickled objects must always be opened in *binary mode* because text files use `str` strings to represent data, not bytes, and the `dump` call simply attempts to write the pickled byte string to an open output file:

```
>>> pickle.dump(['code', 4, '🐍'], open('temp.pkl', 'w')) # bytes+text mode fail
TypeError: write() argument must be str, not bytes

>>> pickle.dump(['code', 4, '🐍'], open('temp.pkl', 'w'), protocol=0)
TypeError: write() argument must be str, not bytes

>>> pickle.dump(['code', 4, '🐍'], open('temp.pkl', 'wb')) # Always binary mode

>>> open('temp.pkl', 'r').read()
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x80 in position 0:...
```

Notice the last result fails here in text mode on macOS; if it doesn't fail for you, it's only because the stored binary data is compatible with your platform's default decoding (i.e., just by luck!). A Windows host, for example, prints gibberish for the last result instead of an error message. Because pickle data is not generally decodable Unicode text, the same rule holds on input as output—correct usage always requires both writing and reading pickle data with binary-mode files, whether unpickling or not:

```
>>> pickle.dump(['code', 4, '🐍'], open('temp.pkl', 'wb')) # Save to file

>>> pickle.load(open('temp.pkl', 'rb'))              # Load from file
['code', 4, '🐍']

>>> open('temp.pkl', 'rb').read()
b'\x80\x04\x95\x15\x00\x00\x00\x00 ...etc... \x04\x8c\x04\xf0\x9f\x90\x8d\x94e.'
```

The Python `json` module, introduced in [Chapter 9](#), is a related tool: it converts a nested-object tree to a text string that can be saved on a file to make it persistent. Unlike `pickle`, `json` doesn't support arbitrary objects (just basic types and built-in containers) but uses a language-neutral scheme that can make it more interoperable with other programs.

Also unlike `pickle`, the `json` module always produces a `str`, so files for saves and loads should generally use *text mode* (JSON files are commonly encoded per UTF-8 for portability, but any encoding works as long as it's consistent and expected):

```
>>> import json
>>> vals = ['code', {'app': ('😄', None, 1.23, 99)}]
>>> json.dumps(vals)
'["code", {"app": ["\ud83d\ude42", null, 1.23, 99]}]'
```

```
>>> text = json.dumps(vals)                          # Save to/load from str
>>> anew = json.loads(text)
>>> anew
['code', {'app': ['😄', None, 1.23, 99]}]
```

```
>>> file = open('data.txt', 'w', encoding='utf8')    # Save/load text-mode file
>>> json.dump(vals, file)
```

```
>>> file.close()
>>> file = open('data.txt', 'r', encoding='utf8')
>>> json.load(file)
['code', {'app': ['😄', None, 1.23, 99]]]
```

Filenames in open and Other Filename Tools

So far in this chapter, we've focused on the *content* of files, but their *names* have a Unicode story too. Its short version is that `str` is the norm for file pathnames in Python and is recommended for portability. If your code, like all the examples so far, uses `str` for filenames and pathnames, they simply work: Python's file tools automatically translate them to and from the encoding used by the host platform and device.

It turns out, though, that Python's file tools also allow you to specify file pathnames as `bytes` to skip automatic filesystem encoding in some contexts. This `bytes` mode may come in handy if you need more control over filename encodings in cross-platform code, but it is rarely needed in typical programs. Per Python's manuals, in fact, this mode need be used only on some Unix systems where undecodable filenames may be present.

Nevertheless, `bytes` filenames do work in these tools, as shown in the following demo, which runs the same on macOS, Windows, Linux, and Android devices tested. As shown, the `open` function happily accepts text or bytes for a file's non-ASCII pathname (this section's examples were run in subfolder `_filenames` in the book's examples package):

```
>>> import sys, os, glob
>>> sys.getfilesystemencoding()          # Filesystem default: macOS and Win11
'utf-8'

>>> name1 = 'hÄck😄1'                    # str filenames
>>> name2 = 'hÄck😄2'
>>> name2.encode('utf8')                 # bytes equivalent for name2 str
b'h\xc3\x84ck\xf0\x9f\x99\x822'

>>> os.listdir()                        # Make files with str and bytes names
[]
>>> open(name1, mode='w', encoding='utf8').write('text1')
5
>>> open(b'h\xc3\x84ck\xf0\x9f\x99\x822', 'w', encoding='utf8').write('text2')
5
>>> os.listdir()                        # Both show up on the filesystem
['hÄck😄1', 'hÄck😄2']

>>> open('hÄck😄2').read()               # And can be accessed either way
'text2'
>>> open('hÄck😄2'.encode('utf8')).read()
'text2'
```

For `bytes` filenames, Python uses UTF-8 encoding on macOS and on Windows since 3.6 (with extra translation for the Windows API's UTF-16). Linux, and by extension Android, accepts any sort of `bytes` for filenames, but Python tries to use UTF-8 when converting to and from `str` (with surrogate escapes for encoding errors: see Python's manuals).

Whether `bytes` filenames are a useful tool or neat parlor trick depends on your use cases, but keep in mind that `bytes` filenames will only work in `open` if their encoding matches the expectations of Python or the underlying filesystem. To demo, the following passes Latin-1 bytes `b'h\xc4ck'` to `open`: this *fails*

on both macOS and Windows as shown (though with a `UnicodeDecodeError` on the latter because it's trapped by Python):

```
>>> 'hÄck'.encode('latin-1'), 'hÄck'.encode('utf-8')
(b'h\xc4ck', b'h\xc3\x84ck')

>>> b'h\xc4ck'.decode('utf8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc4 in position 1:...

>>> open(b'h\xc4ck', 'w')
OSError: [Errno 92] Illegal byte sequence: b'h\xc4ck'

# open(b'h\xc4ck'.decode('latin1'), 'w')    # Works: convert to str
# open(b'h\xc3\x84ck', 'w')              # Works: use UTF-8 encoding
```

By contrast, the preceding code's `open` works on Linux and Android—which later decode the Latin-1 name `b'h\xc4ck'` to `str` by UTF-8, as surrogate-escaped `'h\uddc4ck'`. Still, this works only on drives using a Linux-native *filesystem*. For both its bytes and escaped `str` forms, the Latin-1 name fails in `open` on Linux and Android when the target is a removable drive formatted as exFAT. Hence, the effect of using arbitrary bytes in `open` varies by both platform and filesystem (that is, don't do that!).

The `os` standard-library module's directory-listing `listdir` similarly accepts `str` or bytes and returns a folder's names in the same form, ready to be used in other file tools (per the earlier listing, it also defaults to the current directory if no folder is passed):

```
>>> os.listdir('.')                                # str gives strs, bytes gives bytes
['hÄck👤1', 'hÄck👤2']

>>> os.listdir(b'.')
[b'h\xc3\x84ck\xf0\x9f\x99\x821', b'h\xc3\x84ck\xf0\x9f\x99\x822']

>>> for name in os.listdir('.'):
    print(name, '=>', open(name).read())

hÄck👤1 => text1
hÄck👤2 => text2

>>> for name in os.listdir(b'.'):
    print(name, '=>', open(name).read())

b'h\xc3\x84ck\xf0\x9f\x99\x821' => text1
b'h\xc3\x84ck\xf0\x9f\x99\x822' => text2
```

The `glob` module does filename expansion both ways, too (as `str` or bytes), and `os.fsdecode` decodes filenames per the host's default automatically:

```
>>> glob.glob('h*ck*')                            # str gives str, bytes gives byte
['hÄck👤1', 'hÄck👤2']

>>> glob.glob(b'h*ck*')
[b'h\xc3\x84ck\xf0\x9f\x99\x821', b'h\xc3\x84ck\xf0\x9f\x99\x822']

>>> for name in glob.glob(b'h*ck*'):
    print(name.decode(sys.getfilesystemencoding()), os.fsdecode(name))

hÄck👤1 hÄck👤1
hÄck👤2 hÄck👤2
```

In addition, tools that create and walk folders are similarly flexible for names and paths (demoed on Windows—your path separators and order may vary):

```

>>> os.mkdir('sub🐶') # Make dirs with str or bytes
>>> os.mkdir('sub🐶bytes'.encode('utf8')) # and walk them with both
>>> os.mkdir('sub🐶bytes/subsub👍')

>>> os.listdir()
['hÄck🐶1', 'hÄck🐶2', 'sub🐶', 'sub🐶bytes']
>>> os.listdir('sub🐶bytes')
['subsub👍']

>>> for (dirhere, subshere, filesheere) in os.walk('.'):
    print(dirhere, '=>', subshere, filesheere)

. => ['sub🐶', 'sub🐶bytes'] ['hÄck🐶1', 'hÄck🐶2']
.\sub🐶 => [] []
.\sub🐶bytes => ['subsub👍'] []
.\sub🐶bytes\subsub👍 => [] []
>>>
>>> for (dirhere, subshere, filesheere) in os.walk(b'.'):
    print(dirhere, '=>', subshere, filesheere)

b'.' => [b'sub\x0\x9f\x99\x8a', b'sub\x0\x9f\x99\x8abytes'] [b'...', b'...']
b'.\sub\x0\x9f\x99\x8a' => [] []
b'.\sub\x0\x9f\x99\x8abytes' => [b'subsub\x0\x9f\x91\x8d'] []
b'.\sub\x0\x9f\x99\x8abytes\subsub\x0\x9f\x91\x8d' => [] []

```

While bytes pathnames work as shown and may have some valid roles, it's important to stress that you're almost always better off using `str` strings to name files instead. Doing so leverages tools that already go to great lengths to make pathnames do the right thing and might just avoid at least some portability issues that can arise in apps whose scope must span platforms and devices.

And that's all the time and space we have for this tools survey. Really, Python's standard library and third-party domain are large and evolving toolsets that will likely occupy much of your attention after you've finished this book and move on to real programming tasks. Again, be sure to consult the Python Library Reference soon and often for more info.

Unicode Defaults and UTF-8 Mode

So what's the default encoding, then? This turns out to be a weirdly convoluted story, which we've touched on lightly a few times. Now that we've seen all the players in action, we can finally summarize and finalize this thread.

In short, Python's default Unicode encoding for both *source code* and string encoding/decoding *methods* is always UTF-8 everywhere; for *filenames* can be had with `sys.getfilesystemencoding()`; and for *file content* accessed via `open` is fetched with `locale.getpreferredencoding(False)`.

The *methods* default is `sys.getdefaultencoding()`, but it can no longer be changed as of Python 3.2 (and probably shouldn't have been changed earlier). The *filenames* default is usually UTF-8, including on Windows as of Python 3.6, but is moot if your names are always `str` (as we learned in the preceding section).

The `open` default for *file content* is bifurcated most and badly. Contrary to Python's current docs, it's not simply the result of `locale.getencoding()` on all platforms. Rather, it's either UTF-8 if the *UTF-8 mode* introduced in Python 3.7 is enabled or else `getencoding()`. This matters on Windows

today, where `getencoding()` may be a code-page encoding like `cp1252`; because Unix is usually UTF-8, an open sans encoding isn't portable for non-ASCII content.

UTF-8 mode addresses this, but it must be enabled today by setting the environment variable `PYTHONUTF8` to 1 or using command-line switch `-X utf8`. The result of `getencoding()`, added in Python 3.11, is not influenced by UTF-8 mode the way that `open` and the older `getpreferredencoding(False)` in `locale` are. Hence, `open`'s conditional default.

Both of the `locale` encoding results may also be influenced by platform, environment variables, and command-line switches...*except* on Android, which is just plain UTF-8, and on Windows in the future, when *Python 3.15* will turn UTF-8 mode on by default to match other platforms for code that lacks explicit encodings—a good idea, though too late to the party to avoid making a scene.

Separately, environment variable `PYTHONIOENCODING` can be used to give the encoding of *stdio streams* (e.g., `sys.stdout`) when they are redirected to files on Windows and others (e.g., `> output.txt`)...*except* when you or a Python of the future enable 3.7's UTF-8 mode, which applies to redirected streams too...*unless* variable `PYTHONIOENCODING` is also set.

Beyond all this, the encoding fate of unredirected console streams on Windows can be sealed with `PYTHONLEGACYWINDOWSSTDIO`; filename encodings on Windows may also backslide to their code-page roots for the brazenly grandiose `PYTHONLEGACYWINDOWSFSENCODING`; Unix encodings can be forced to skip UTF-8 similarly with `PYTHONCOERCECLOCALE`; and UTF-8 mode modulates additional textual tools omitted here for space (and humanity).

Right. If you don't care to remember all that—and prefer to sleep well at night—enable UTF-8 mode on Windows today before Python 3.15 does; use `str` for filenames and pass explicit encodings to `open` instead of relying on tangled and morphing defaults; and see Python's manuals for the full, if frightening, story.

The Unicode Twilight Zone

To wrap up, this chapter is going to briefly present two curated topics from the Unicode realm—BOMs and normalization—that are too convoluted for full coverage in this book and probably too arcane to matter to most Python newcomers. If and when these become important to your projects, you'll find ample resources in both Python's docs and the web at large to fill in the bits glossed over here for space. For now, let's jump right into the first of these arguably explosive topics.

Dropping the BOM in Python

As noted briefly earlier in this chapter, some encoding schemes store a special *byte order marker* (BOM) sequence at the start of files to specify data endianness (which end of a string of bits is most significant to its value) or declare the encoding type in general. Python's text-mode files both skip this marker on input and write it on output if the encoding implies presence, but we sometimes must use a specific encoding name to force BOM processing explicitly and may need to accommodate it when encoding manually.

For example, in the UTF-16 and UTF-32 encodings, the BOM both identifies the encoding and specifies big- or little-endian format. A UTF-8 text file may also include a BOM, but this isn't guaranteed and serves only to declare that it is UTF-8 in general.

When reading and writing data using these encoding schemes, Python automatically skips or writes the BOM if it is either implied by a general encoding name or if you provide a more specific encoding name to force the issue. More concretely:

- In *UTF-16*, the BOM is always processed for encoding name `utf-16`, and the more specific encoding name `utf-16-le` denotes little-endian format.
- In *UTF-8*, the more specific encoding name `utf-8-sig` forces Python to both skip and write a BOM on input and output, respectively, but the general `utf-8` does not.

Making BOMs in Text Editors

Let's make some files with BOMs to see how this works in practice. We're going to do this in Python in a moment, too, but let's start by creating a file in a text editor to underscore that your programs will also have to process content from other sources. If you wish to work along, you can use any text editor that's Unicode aware for saves, and most editors on PCs, tablets, and phones are today. This demo uses Windows Notepad just because it's widespread (and documenting multiple editors' usage here is right out).

When you save a text file in Windows Notepad, you can specify its encoding type in a drop-down list—simple text, little- or big-endian UTF-16, or UTF-8 with or without a BOM. For instance, if you use Notepad to save the two lines “code” and “CODE” in a text file named *code.txt* with encoding type *ANSI*, the file is written as simple ASCII text without a BOM. Technically, the save uses the cp1252 default encoding on the Windows host used, but cp1252 is an ASCII superset, and the content is all ASCII.

In Python after the save, when this file is read in binary mode, we can see the actual bytes stored in the file, including `\r` in newlines. When it's read as text, Python performs newline translation by default, and we can also decode it explicitly as UTF-8 text since ASCII is a subset of both this encoding and Windows' default cp1252:

```
$ py -3                                     # On Windows, post ANSI save in Notepad
>>> import locale
>>> locale.getpreferredencoding(False)      # open default: ASCII&Latin-1 superset
'cp1252'
>>> open('code.txt', 'rb').read()          # ASCII (and cp1252 and UTF-8) bytes
b'code\r\nCODE\r\n'
>>> open('code.txt', 'r').read()           # Text mode also translates newlines
'code\nCODE\n'
>>> open('code.txt', 'r', encoding='utf-8').read()
'code\nCODE\n'
```

Now, if this file is instead saved as *UTF-8 with BOM* in Notepad (*UTF* in its prior versions), its text is prepended with a three-byte UTF-8 BOM sequence (which prints as non-ASCII `\xNN` hex escapes in bytes in Python), and we need to give the more specific encoding name `utf-8-sig` to force Python to skip the marker automatically on input (else it prints as a `\uNNNN` Unicode code-point escapes in `str`):

```
>>> open('code.txt', 'rb').read()          # After resaved: UTF-8 with 3-byte BOM
b'\xef\xbb\xbfcode\r\nCODE\r\n'
>>> open('code.txt', 'r').read()           # Default+utf8 keep BOM, utf-8-sig drops
'ï»¿code\nCODE\n'
>>> open('code.txt', 'r', encoding='utf-8').read()
'\ufeffcode\nCODE\n'
>>> open('code.txt', 'r', encoding='utf-8-sig').read()
'code\nCODE\n'
```

And if the file is stored as *UTF-16 BE* in Notepad (*Unicode big endian* formerly), we get UTF-16 big-endian format data in the file, with two bytes (16 bits) prepended to record a two-byte BOM sequence. The encoding name `utf-16` in Python skips the BOM because it is implied (since all UTF-16 files have a BOM), and `utf-16-be` handles the big-endian format but does not skip the BOM in the input result:

```
>>> open('code.txt', 'rb').read()
b'\xfe\xff\x00c\x00o\x00d\x00e\x00r\x00n\x00C\x00O\x00D\x00E\x00r\x00n'

>>> open('code.txt', 'r', encoding='utf-16').read()
'code\nCODE\n'

>>> open('code.txt', 'r', encoding='utf-16-be').read()
'\uffeffcode\nCODE\n'
```

Experiment with other save/open combinations for more insights. The default encoding in the last example, for example, would print garbage characters because it's not valid for the text, and UTF-16 little-endian swaps byte order for the BOM and each 2-byte character.



Notepad flux: Notepad recently changed its encoding options for saves. Today, it offers ANSI (the host's default), UTF-16 in little- and big-endian flavors, and UTF-8 with and without a BOM, and defaults to UTF-8. When this book's prior edition was penned, Notepad had UTF-8 with an implied BOM, and its “Unicode” meant UTF-16—which, of course, is just one of the very many kinds of Unicode encoding. The narrative here has been updated to reflect the new choices, but this naturally is just the story today. Because this book staunchly refuses to become a Notepad doc, translate save options as needed to the Notepad on a PC near you.

Making BOMs in Python

The preceding section used Python to read files made in an editor, but the same patterns apply when Python also makes the files: Python will automatically add a BOM when we write a Unicode file in code, but we need to use a more explicit encoding name to force the BOM in UTF-8 mode—`utf-8` does not write the BOM on output but `utf-8-sig` does, and the same goes for skipping UTF-8 BOMs on input. When run on Windows (Unix is the same, but doesn't add `\r` to newlines, and usually defaults to UTF-8):

```
>>> open('temp.txt', 'w', encoding='utf-8').write('code\nCODE\n')
10
>>> open('temp.txt', 'rb').read()                                     # utf-8: no BOM
b'code\r\nCODE\r\n'

>>> open('temp.txt', 'w', encoding='utf-8-sig').write('code\nCODE\n')
10
>>> open('temp.txt', 'rb').read()                                     # utf-8-sig: adds BOM
b'\xef\xbb\xbfcode\r\nCODE\r\n'

>>> open('temp.txt', 'r').read()                                       # Default: bad BOM
'i»¿code\nCODE\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()                   # utf-8: keeps BOM
'\uffeffcode\nCODE\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()               # utf-8-sig: drops BOM
'code\nCODE\n'
```

Per this code, although utf-8 does not drop the BOM when one is present, data *without* a BOM can be read with both utf-8 and utf-8-sig—which means you can use the latter for input if you’re not sure whether a BOM is present in a file or not (and don’t read this paragraph out loud in an airport security line!):

```
>>> open('temp.txt', 'w').write('code\nCODE\n')          # Default: UTF-8 subset
10
>>> open('temp.txt', 'rb').read()                          # No BOM
b'code\r\nCODE\r\n'

>>> open('temp.txt', 'r').read()                            # Default
'code\nCODE\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()          # Either utf-8 works
'code\nCODE\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()
'code\nCODE\n'
```

For the encoding name utf-16, the BOM is handled automatically: on *output*, data is written in the platform’s native endianness, and the BOM is always written; on *input*, data is decoded per the BOM, and the BOM is always stripped. This reflects the fact that BOMs are standard and required in the UTF-16 encoding scheme:

```
>>> import sys
>>> sys.byteorder          # Windows host's default endianness
'little'
>>> open('temp.txt', 'w', encoding='utf-16').write('code\nCODE\n')
10

>>> open('temp.txt', 'rb').read()
b'\xff\xfec\x00o\x00d\x00e\x00r\x00\n\x00C\x00O\x00D\x00E\x00r\x00\n\x00'

>>> open('temp.txt', 'r', encoding='utf-16').read()
'code\nCODE\n'
```

More specific UTF-16 encoding names can specify different endianness, though you may have to manually write and skip the BOM yourself in some scenarios if it is required or present—study the following examples for more BOM-making instructions (sorry):

```
>>> open('temp.txt', 'r', encoding='utf-16-le').read()
'\uffeffcode\nCODE\n'

>>> open('temp.txt', 'w', encoding='utf-16-be').write('\uffeffcode\nCODE\n')
11
>>> open('temp.txt', 'rb').read()
b'\xfe\xff\x00c\x00o\x00d\x00e\x00r\x00\n\x00C\x00O\x00D\x00E\x00r\x00\n'

>>> open('temp.txt', 'r', encoding='utf-16').read()
'code\nCODE\n'
>>> open('temp.txt', 'r', encoding='utf-16-be').read()
'\uffeffcode\nCODE\n'
```

The more specific UTF-16 encoding names by themselves create and work fine with BOM-less files, though utf-16 requires one on input in order to determine byte order:

```
>>> open('temp.txt', 'w', encoding='utf-16-le').write('CODE')
4
>>> open('temp.txt', 'rb').read()          # Okay if BOM not present or expected
b'C\x00O\x00D\x00E\x00'

>>> open('temp.txt', 'r', encoding='utf-16-le').read()
'CODE'
>>> open('temp.txt', 'r', encoding='utf-16').read()
UnicodeError: UTF-16 stream does not start with BOM
```

Experiment with these encodings yourself, or see Python's library manuals for more details on the BOM. And if you really want to drop the bomb in Python (and stretch this section's silly bit to its breaking point), Unicode emoji characters do the job:

```
>>> open('boms.txt', 'w', encoding='utf-8-sig').write('💣' * 10)
10
>>> open('boms.txt', encoding='utf-8-sig').read()
'💣💣💣💣💣💣💣💣💣💣'
```

Unicode Normalization: Whither Standard?

Last but not least, after devoting dozens of pages to Unicode, we'll close by explaining one way in which it, at least arguably, falls short. In brief, this standard *failed to standardize* code points for a handful of characters: it allows the same character to be represented in more than one way, which breaks equality testing and can wreak interoperability havoc with text-processing programs. While there may have been valid rationales for this policy, it adds a special case for programmers and undoubtedly breaks many a tool and app.

For example, the character ñ (an n augmented with a tilde, commonly used in Spanish) can be represented with two different code-point sequences in the Unicode standard:

- As a single character with code point `\u00F1`—an augmented n
- As a two-codepoint sequence `\u006E` and `\u0303`—a naked n followed by a combining tilde

The first of these is called the *composed* form, known as *NFC*, because it combines letter and accent. The second is called the *decomposed* form, known as *NFD*, because its parts are split. Despite their glaring difference, the Unicode standard mandates that these two forms represent the same character ñ and must be treated as such by programs.

It's easy to observe these alter egos in Python: the following uses `\u...` Unicode code-point escapes to specify the code points for ñ in both forms (Windows users: don't be alarmed if some characters, especially NFDs, render differently in command-line interfaces due to settings that are well beyond this book's scope):

```
>>> L = '\u00F1'          # NFC form ('xF1' works too)
>>> M = '\u006E\u0303'    # NFD form
>>> L, M                  # The same character
('ñ', 'ñ')
```

Nor is this character alone in its split personality. Other characters such as Å, è, é, and ÿ have both NFC and NFD representations as well:

```
>>> '\u00C5', '\u0041\u030A'    # NFD (1 code point), NFC (2 code points)
('Å', 'Å')
>>> '\u00E8', '\u0065\u0300'    # But it's the same character for both
('è', 'è')
>>> '\u00E9', '\u0065\u0301'
('é', 'é')
>>> '\u00D4', '\u003D\u0308'
('ÿ', 'ÿ')
```

Importantly, this is not about the *encoded* representation of this character in bytes, which naturally varies across different encodings. The two alternative representations for each preceding character differ for their decoded, in-memory representation as integer *code points*. Encoded forms (e.g., stored in files) differ, too, but decode to the same differing code points:

```
>>> '\u00F1'.encode('utf8'), '\u006E\u0303'.encode('utf8')
(b'\xc3\xbf', b'n\xcc\x83')

>>> b'\xc3\xbf'.decode('utf8'), b'n\xcc\x83'.decode('utf8')
('ñ', 'ñ')
```

The unfortunate consequence of this plurality is that it *breaks equality testing*: because the same character can be represented in multiple ways, it's impossible to compare with the usual tools. In Python specifically, the `==` equal-by-value operator, which compares characters (really, code points), won't suffice in the presence of Unicode doppelgängers:

```
>>> L = '\u00F1'
>>> M = '\u006E\u0303'

>>> L, M                                # The same character
('ñ', 'ñ')

>>> L == M                              # But equality says no!
False

>>> len(L), len(M)                      # Because their code points differ
(1, 2)
```

All of which might not be a problem in an ideal computing world that settled on one form as a de facto standard for portability's sake. Unfortunately, that's not the world we occupy. Computer vendors, being computer vendors, have opted to favor different forms: broadly speaking, macOS prefers NFD, Windows and others prefer NFC, this can also vary by filesystem, and tolerance of nonnative forms is less than complete. The net effect is that the Unicode standard created an interoperability problem while trying to fix another!

So what to do with a world that just won't standardize? The trick is that, for every tool that processes file contents or names across divergent platforms, text must be converted to a common form before comparisons. And luckily, Python makes this remarkably easy:

```
>>> from unicodedata import normalize    # Python stdlib tool
>>> L = '\u00F1'
>>> M = '\u006E\u0303'
>>> L, M                                # Same char, diff code points
('ñ', 'ñ')

>>> L == M                              # Equality fails
False

>>> normalize('NFC', L) == normalize('NFC', M)  # Common-form equality works
True
>>> normalize('NFD', L) == normalize('NFD', M)  # Either suffices, if the same
True
```

Programs that compare filenames sent between platforms, for example, should be careful to normalize this way. Otherwise, files whose names look the same to users (and really *are* the same per Unicode) will fail to match by simple equality and derail searches and syncs. The same goes for text files obtained from arbitrary sources—like most internet content.

There's more to this story (e.g., the *canonical* equivalence of NFC and NFD normalized forms is still not the same as *compatibility*, though most programs don't need to care). Because we've run tight on space, though, we'll stop short. If you'd like to dig deeper, you'll find ample follow-up coverage both on the web and in Python's manual set (see the latter's [Unicode HOWTO](#) and its coverage of the related string `casefold` method).

At the least, though, you now shouldn't be wholly surprised when your text-processing code is bitten by this curious choice of the Unicode standard.

Chapter Summary

This chapter explored the advanced string support available in Python for processing Unicode text and binary data. While some programmers use ASCII text and may get by with the basic string type and its operations, Python's string model fully supports both richer Unicode text via the normal `str` string type and byte-oriented binary data with `bytes` and `bytearray`.

In addition, we learned how Python's file object automatically encodes and decodes Unicode text, and deals with byte strings for binary-mode files. Finally, we briefly met some text and binary tools in Python's library and sampled their behavior with strings, and took a look at the darker Unicode corners of BOMs and normalization.

In the next chapter, we'll shift our focus to tool-builder topics, with a survey of ways to manage access to object attributes by inserting automatically run code. Before we move on, though, here's a set of questions to review what we've learned. This has been a substantial chapter, so be sure to read the quiz answers eventually for a more in-depth summary.

Test Your Knowledge: Quiz

1. What are the names and roles of string object types in Python?
2. How do Python's string types differ in terms of operations?
3. How can you code non-ASCII Unicode characters in a string in Python?
4. What are the main differences between text- and binary-mode files in Python?
5. How would you read a Unicode text file that contains text in a different encoding than the default for your platform?
6. How can you create a Unicode text file in a specific encoding format?
7. Why is ASCII text considered to be a kind of Unicode text?
8. What do BOM and normalization mean in Unicode?
9. How large an impact does Python's text/binary string dichotomy have on your code?

Test Your Knowledge: Answers

1. Python has three string types: `str` (for Unicode text, including ASCII), `bytes` (for binary data with absolute byte values), and `bytearray` (a mutable flavor of bytes). The `str` type usually represents content stored in text files, and the other two types generally represent content stored in binary files (including encoded text).
2. Python's string types share almost all the same operations: method calls, sequence operations, and even larger tools like pattern matching work the same way. On the other hand, only `str` supports string formatting's `format` method and `f'...'` f-strings, and `bytearray` has an additional set of operations that perform in-place changes. The `str` and `bytes` types also have methods for encoding and decoding text, respectively.

3. Non-ASCII Unicode characters can be coded in a `str` string with both hex (`\xNN`) and Unicode (`\UNNNNN`, `\UNNNNNNNNN`) escapes for code points, both of which denote character code points. They can also be coded in their encoded form as `bytes` using hex escapes and decoded to text. On most devices, non-ASCII characters—accented characters and emojis, for example—can also be typed or pasted directly into code and are interpreted per the UTF-8 default or an encoding-directive comment at the top of a source code file.
4. Text-mode files assume their content is Unicode text (even if it's all ASCII) and automatically decode when reading and encode when writing. With binary-mode files, bytes are transferred to and from the file unchanged. The contents of text-mode files are usually represented as `str` objects in your script, and the contents of binary files are represented as `bytes` (or `bytearray`) objects. Text-mode files also handle BOMs for certain encoding types and automatically translate newline sequences to and from the single `\n` character on input and output unless this is explicitly disabled; binary-mode files do not perform either of these steps.
5. To read files encoded in a different encoding than the default for your platform, simply pass the name of the file's encoding to the `open` built-in function; data will be decoded per the specified encoding when it is read from the file, and you'll get back a decoded Unicode-text `str` string that has no encoding. You can also read in binary mode and manually decode the bytes to a string by giving an encoding name, but this involves extra work and is somewhat error-prone for multibyte characters (you may accidentally read a partial character sequence when loading content by bytes or chunks).
6. To create a Unicode text file in a specific encoding format, pass the desired encoding name to `open`; strings will be encoded per the desired encoding when they are written to the file. You can also manually encode a string to bytes and write it in binary mode, but this is usually extra work.
7. ASCII text is considered to be a kind of Unicode text because its 7-bit (0..127) range of values is a subset of many Unicode encodings. For example, valid ASCII text is also valid Latin-1 text (Latin-1 simply assigns the remaining possible values in an 8-bit byte to additional characters) and valid UTF-8 text (UTF-8 uses a variable-byte scheme for representing more characters, but ASCII characters are still represented with the same values in a single byte). This makes Unicode backward compatible with ASCII, as long as the encodings used represent ASCII the same way.
8. The Unicode BOM is a sequence of bytes added to the front of a text string or file in some encodings to both identify the encoding and give the string's endianness. Unicode normalization converts characters to a common format to neutralize differences that arise for characters that have multiple code-point values in the Unicode standard and would fail to match by simple code-point (character) equality.
9. The impact of Python's strings model depends upon the types of strings you use. For scripts that use simple ASCII text on platforms with ASCII-compatible default encodings, the impact is probably minor: the `str` string sans encodings suffices in this case. Moreover, although string-related tools in the standard library, such as `re`, `struct`, `pickle`, and `os`, may technically use different types in different contexts, the effect is largely irrelevant to most programs because Python's `str` and `bytes` support almost identical interfaces. On the other hand, if you process non-ASCII Unicode text, you'll need to use `str` and pass encodings to `open`; if you deal with binary data files, you'll need to deal with content as `bytes` objects; and if your code must work across many platforms or process content from arbitrary sources, you'll want to use explicit encodings for text files. In general, Unicode is the way the text world works today and will be a must-know tool for most Python users.

Managed Attributes

This chapter expands on the *attribute interception* techniques introduced earlier, introduces another, and employs them in a handful of larger examples. Like everything in this part of the book, this chapter is classified as an advanced topic and optional reading, because most applications programmers don't need to care about the material discussed here—they can fetch and set attributes on objects without concern for attribute implementations.

Especially for tools builders, though, managing attribute access can be an important part of flexible APIs. Moreover, an understanding of the descriptor model covered here can make related tools such as slots and properties more tangible and may even be required reading if it appears in code you must use.

Why Manage Attributes?

Object attributes are central to most Python programs—they are where we often store information about the entities our scripts process. Normally, attributes are simply names for objects; a person's `name` attribute, for example, might be a simple string, fetched and set with basic attribute syntax:

```
person.name           # Fetch attribute value
person.name = value   # Change attribute value
```

In most cases, the attribute lives in the object itself or is inherited from a class from which it derives. That basic model suffices for most programs you will write in your Python career.

Sometimes, though, more flexibility is required. Suppose you've written a program to use a `name` attribute directly, but then your requirements change—for example, you decide that names must be validated or mutated with program logic when accessed. It's straightforward to code methods to manage access to the attribute's value (`valid` and `transform` are abstract and hypothetical here):

```

class Person:
    def getName(self):
        if not valid():
            raise TypeError('cannot fetch name')
        else:
            return self.name.transform()

    def setName(self, value):
        if not valid(value):
            raise TypeError('cannot change name')
        else:
            self.name = transform(value)

person = Person()
person.getName()
person.setName('value')

```

The problem with this is that it also requires changing all the places where names are used in the entire program—a possibly nontrivial task. Moreover, this approach requires the program to be aware of how values are exported: as simple names or called methods. If you begin with a method-based interface to data, clients are immune to changes; if you do not, changes can become problematic.

This issue can crop up more often than you might expect. The value of a cell in a spreadsheet-like program, for instance, might begin its life as a simple discrete value but later mutate into an arbitrary calculation. Since an object's interface should be flexible enough to support such future changes without breaking existing code, switching to methods later is less than ideal.

Inserting Code to Run on Attribute Access

A better solution would allow you to run code automatically on attribute access if needed. That's one of the main roles of managed attributes—they provide ways to add *attribute accessor* logic after the fact. More generally, they support arbitrary attribute usage modes that go beyond simple data storage.

At various points in this book, we've met Python tools that allow our scripts to dynamically compute attribute values when fetching them and validate or change attribute values when storing them. In this chapter, we're going to focus more deeply on the tools already introduced, explore other tools in this category, and study some larger use-case examples in this domain. Specifically, this chapter presents *four* accessor techniques:

1. The property built-in, for specifying methods to handle access to a specific attribute
2. The `__get__` and `__set__` descriptor methods, for handling access to a specific attribute and the basis for other tools such as properties and slots
3. The `__getattr__` and `__setattr__` methods, for handling undefined attribute fetches and all attribute assignments
4. The `__getattribute__` method, for handling all attribute fetches

We met these tools in Chapters 30 and 32 briefly, and in some cases, hardly at all. As you'll see here, all four techniques share goals to some degree, and it's usually possible to code a given problem using any one of them.

That said, they also differ in some important ways. For example, the last two techniques listed here apply to *specific* attributes, whereas the first two are generic enough to be used by delegation-based proxy classes that must route *arbitrary* attributes to wrapped objects. As you'll find, all four schemes also differ in both complexity and aesthetics in ways you must see in action to judge for yourself.

Besides studying the specifics behind these four attribute interception techniques, this chapter also presents an opportunity to explore programs larger than most we've seen elsewhere in this book. The `CardHolder` case study at the end, for example, should serve as a self-study example of larger classes in action. We'll also be using some of the techniques outlined here in the next chapter to code decorators, so be sure you have at least a general understanding of these topics before you move on.

Properties

Up first, the property protocol allows us to route a specific attribute's get, set, and delete operations to functions or methods we provide, enabling us to insert code to be run automatically on attribute accesses, intercept attribute deletions, and provide documentation for attributes if desired.

As introduced in [Chapter 32](#), properties are created with the `property` built-in and are assigned to class attributes, just like method functions. Accordingly, they are inherited by subclasses and instances, like any other class attributes. Their access-interception functions are provided with the `self` instance argument, which grants access to state information and class attributes available on the subject instance.

A property manages a single, specific attribute; although it can't catch all attribute accesses generically, it allows us to control both fetch and assignment accesses and enables us to change an attribute from simple data to a computation freely without breaking existing code. As you'll see, properties are strongly related to descriptors; in fact, they are essentially a restricted form of them.

The Basics

A property is created by assigning the result of a built-in function to a class attribute:

```
attribute = property(fget, fset, fdel, doc)
```

None of this built-in's arguments are required, and all default to `None` if not passed. For the first three, this `None` means that the corresponding operation is not supported, and attempting it will raise an `AttributeError` exception automatically.

When these arguments are used, we pass `fget` a function for intercepting attribute fetches, `fset` a function for assignments, and `fdel` a function for attribute deletions. Technically, all three of these arguments accept any callable, including a class's method, having a first argument to receive the instance being qualified. When later invoked, the `fget` function returns the computed attribute value, `fset` and `fdel` return nothing (really, `None`), and all three may raise exceptions to reject access requests.

The `doc` argument receives a documentation string for the attribute if desired. If omitted, the property copies the docstring of the `fget` function, which, as usual, defaults to `None`.

This built-in property call returns a property object, which we assign to the name of the attribute to be managed in the class scope, where it will be inherited by every instance. As you'll learn ahead, this assignment can be automated by `@` decorator syntax, though its distributed usage may seem awkward for set and delete methods. However assigned, later accesses to the attribute automatically invoke the property's handlers.

A First Example

To demonstrate how this translates to working code, the class in [Example 38-1](#) uses a property to trace access to an attribute named `name`; the actual stored data is named `_name` so it does not clash with the property.

Example 38-1. prop-person.py

```
class Person:
    def __init__(self, name):
        self._name = name

    def getName(self):
        print('fetch...')
        return self._name

    def setName(self, value):
        print('change...')
        self._name = value

    def delName(self):
        print('remove...')
        del self._name

    name = property(getName, setName, delName, 'name property docs')

sue = Person('Sue Jones')           # sue has a managed attribute
print(sue.name)                     # Runs getName
sue.name = 'Susan Jones'            # Runs setName
print(sue.name)
del sue.name                         # Runs delName

print('-'*20)
bob = Person('Bob Smith')           # bob inherits property too
print(bob.name)
print(Person.name.__doc__)           # Or help(Person.name)
```

This particular property doesn't do much—it simply intercepts and traces an attribute—but it serves to demonstrate the protocol. When this code is run, two instances inherit the property, just as they would any other attribute attached to their class. However, accesses to their `name` attribute are caught and managed by the code we provide:

```
$ python3 prop-person.py
fetch...
Sue Jones
change...
fetch...
Susan Jones
remove...
-----
fetch...
Bob Smith
name property docs
```

Like all class attributes, properties are *inherited* by both instances and lower subclasses. If we change our example as follows, for instance:

```

class Super:
    ...the original Person class code...
    name = property(getName, setName, delName, 'name property docs')

class Person(Super):
    pass                                # Properties are inherited (class attrs)

sue = Person('Sue Jones')
...rest unchanged...

```

the output is the same—the `Person` subclass inherits the `name` property from `Super`, and the `sue` instance gets it from `Person`. In terms of inheritance, properties work the same as normal methods; because they have access to the `self` instance argument, they can access instance state information and methods irrespective of subclass depth, as the next section further demonstrates.

Computed Attributes

The example in the prior section simply traces attribute accesses. Usually, though, properties do much more—computing the value of an attribute dynamically when fetched, for instance, as [Example 38-2](#) illustrates.

Example 38-2. prop-computed.py

```

class PropSquare:
    def __init__(self, start):
        self.value = start

    def getX(self):
        return self.value ** 2          # On attr fetch

    def setX(self, value):
        self.value = value             # On attr assign

    X = property(getX, setX)            # No delete or docs

P = PropSquare(3)                      # Two instances of class with property
Q = PropSquare(32)                     # Each has different state information

print(P.X)                             # 3 ** 2
P.X = 4
print(P.X)                             # 4 ** 2
print(Q.X)                             # 32 ** 2 (1024)

```

This class defines an attribute `X` that is accessed as though it were simple data, but really runs code to compute its value when fetched. The net effect triggers an implicit method call. When the code is run, the value is stored in the instance as state information, but each time we fetch it via the managed attribute, its value is automatically squared:

```

$ python3 prop-computed.py
9
16
1024

```

Notice that we’ve made two different instances—because property methods automatically receive a `self` argument, they have access to the state information stored in instances. In our case, this means the fetch computes the square of the subject instance’s own data.

Coding Properties with Decorators

Although we're saving additional details until the next chapter, we introduced *function decorator* basics earlier, in [Chapter 32](#). Recall that the function decorator syntax:

```
@decorator
def func(args): ...
```

is automatically translated to this equivalent by Python to *rebind* the function name to the result of the decorator callable:

```
def func(args): ...
func = decorator(func)
```

Because of this mapping, the property built-in can automatically serve as a decorator to define a function that will run automatically when an attribute is fetched:

```
class Person:
    @property
    def name(self): ...           # Rebinds: name = property(name)
```

When run, the decorated method is automatically passed to the first argument of the property built-in. This is really just alternative syntax for creating a property and rebinding the attribute name manually, but may be seen as more explicit in this role:

```
class Person:
    def name(self): ...
    name = property(name)       # Manual equivalent to @property
```

Setter and deleter decorators

The preceding works naturally for property get functions, but what about other accesses? In full detail, property objects also have getter, setter, and deleter methods that assign the corresponding property accessor methods and return a copy of the property itself. We can use these to specify components of properties by decorating normal methods, too, though the getter component (along with attributes docs) is usually filled in automatically by the act of creating the property itself. [Example 38-3](#) demos the basics.

Example 38-3. *prop-person-deco.py*

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):           # name = property(name)
        'name property docs'
        print('fetch...')
        return self._name

    @name.setter
    def name(self, value):    # name = name.setter(name)
        print('change...')
        self._name = value

    @name.deleter
    def name(self):          # name = name.deleter(name)
        print('remove...')
        del self._name
```

```

sue = Person('Sue Jones')           # sue has a managed attribute
print(sue.name)                     # Runs name getter (def name 1)
sue.name = 'Susan Jones'           # Runs name setter (def name 2)
print(sue.name)
del sue.name                         # Runs name deleter (def name 3)

print('-'*20)
bob = Person('Bob Smith')           # bob inherits property too
print(bob.name)
print(Person.name.__doc__)           # Or help(Person.name)

```

In fact, this code is equivalent to the first example in this section—decoration is just an alternative way to code properties in this case. When it's run, the results are the same:

```

$ python3 prop-person-deco.py
fetch...
Sue Jones
change...
fetch...
Susan Jones
remove...
-----
fetch...
Bob Smith
name property docs

```

Compared to manual assignment of property results, using decorators to properties in this example requires just three extra lines of code—a seemingly negligible difference. As is so often the case with alternative tools, though, the choice between the two techniques is largely subjective.

Descriptors

Very briefly previewed in [Chapter 32](#), *descriptors* provide an alternative way to intercept attribute access; they are strongly related to the properties discussed in the prior section. Really, a property is a kind of descriptor—technically speaking, the property built-in is just a simplified way to create a specific type of descriptor that runs method functions on attribute accesses. In fact, descriptors are the underlying implementation mechanism for a variety of class tools, including both properties and slots, and play other internal roles in Python that we can safely skip here.

Functionally speaking, the descriptor protocol allows us to route a specific attribute's get, set, and delete operations to methods of a separate class's instance object that we provide. This allows us to insert code to be run automatically on attribute fetches and assignments, intercept attribute deletions, and provide documentation for the attributes if desired.

Descriptors are created as independent *classes*, and they are assigned to class attributes just like method functions. Like any other class attribute, they are inherited by subclasses and instances. Their access-interception methods are provided with both a `self` for the descriptor instance itself as well as the instance of the client class whose attribute references the descriptor object. Because of this, they can retain and use state information of their own, as well as state information of the subject instance. For example, a descriptor may call methods available in the client class, as well as descriptor-specific methods it defines.

Like a property, a descriptor manages a single, specific attribute; although it can't catch all attribute accesses generically, it provides control over both fetch and assignment accesses and allows us to change an attribute name freely from simple data to a computation without breaking existing code. If

this sounds like properties, it's because it is: as you shall see, properties can be coded as descriptors directly.

Unlike properties, though, descriptors provide a more general tool. For instance, because they are coded as normal classes, descriptors have their own state, may participate in descriptor inheritance hierarchies, can use composition to aggregate objects, and provide a natural structure for coding internal methods and attribute documentation strings.

The Basics

As mentioned, descriptors are coded as separate classes and provide specially named accessor methods for the attribute access operations they wish to intercept—get, set, and deletion methods in the descriptor class are automatically run when the attribute assigned to the descriptor class instance is accessed in the corresponding way:

```
class Descriptor:
    "docstring goes here"
    def __get__(self, instance, owner): ...      # Return attr value
    def __set__(self, instance, value): ...      # Return nothing (None)
    def __delete__(self, instance): ...          # Return nothing (None)
```

Classes with any of these methods are considered descriptors, and their methods are special when one of their instances is assigned to another class's attribute—when the attribute is accessed, these methods are automatically invoked.

If any of these methods are absent, it generally means that the corresponding type of access is not supported. Unlike properties, however, omitting a `__set__` allows the descriptor attribute's name to be assigned and thus redefined in an instance, thereby *hiding* the descriptor—to make an attribute *read-only*, you must define `__set__` to catch assignments and raise an exception.

Descriptors with `__set__` methods also have some special-case implications for inheritance that we'll largely defer until [Chapter 40](#)'s coverage of metaclasses and the complete inheritance specification. In short, a descriptor with a `__set__` is known formally as a *data descriptor* and is given precedence over other names located by normal inheritance rules. The inherited descriptor for attribute `__class__`, for example, overrides the same name in an instance's namespace dictionary. This also works to ensure that data descriptors you code in your own classes take precedence over others.

Descriptor method arguments

Before we code anything realistic, let's take a brief look at some fundamentals. All three descriptor methods outlined in the prior section are passed both the descriptor class instance (`self`) and the instance of the client class to which the descriptor instance is attached (`instance`).

The `__get__` access method additionally receives an `owner` argument, specifying the class to which the descriptor instance is attached. Its `instance` argument is either the instance through which the attribute was accessed (for `instance.attr`), or `None` when the attribute is accessed through the owner class directly (for `class.attr`). The former of these generally computes a value for instance access, and the latter usually returns `self` if descriptor object access is supported.

For example, in the following REPL session, when `X.attr` is fetched, Python automatically runs the `__get__` method of the `Descriptor` class instance to which the `Subject.attr` class attribute is assigned:


```

>>> class Descriptor:
    def __get__(self, instance, owner):
        print(self, instance, owner, sep='\n')

>>> class Subject:
    attr = Descriptor()           # Descriptor instance is class attr

>>> X = Subject()
>>> X.attr
<__main__.Descriptor object at 0x104bc9b20>
<__main__.Subject object at 0x104b8a570>
<class '__main__.Subject'>

>>> Subject.attr
<__main__.Descriptor object at 0x104bc9b20>
None
<class '__main__.Subject'>

```

Notice the arguments automatically passed in to the `__get__` method in the first attribute fetch—when `X.attr` is fetched, it's as though the following translation occurs (though the `Subject.attr` here doesn't invoke `__get__` again as it normally would):

```
X.attr => Descriptor.__get__(Subject.attr, X, Subject)
```

The descriptor knows it is being accessed directly when its instance argument is `None`.

Read-only descriptors

As mentioned earlier, unlike properties, simply omitting the `__set__` method in a descriptor isn't enough to make an attribute read-only because the descriptor name can be assigned in an instance. In the following, the attribute assignment to `X.a` stores `a` in the instance object `X`, thereby hiding the descriptor stored in class `C`:

```

>>> class D:
    def __get__(*args): print('get')

>>> class C:
    a = D()           # Attribute "a" is a descriptor instance

>>> X = C()
>>> X.a              # Runs inherited descriptor __get__
get
>>> C.a
get
>>> X.a = 99         # Stored on X, hiding C.a!
>>> X.a
99
>>> list(X.__dict__.keys())
['a']
>>> Y = C()
>>> Y.a              # Y still inherits descriptor
get
>>> C.a
get

```

This is the way all instance attribute assignments work in Python, and it allows classes to selectively override class-level defaults in their instances. To make a descriptor-based attribute read-only, catch the assignment in the descriptor class and raise an exception to prevent attribute assignment—when

assigning an attribute that is a descriptor, Python effectively bypasses the normal instance-level assignment behavior and routes the operation to the descriptor object:

```
>>> class D:
    def __get__(*args): print('get')
    def __set__(*args): raise AttributeError('cannot set')

>>> class C:
    a = D()

>>> X = C()
>>> X.a                                     # Routed to C.a.__get__
get
>>> X.a = 99                               # Routed to C.a.__set__
AttributeError: cannot set
```



The deletion trio: Be careful not to confuse the descriptor `__delete__` method with the general `__del__` method. The former is called on attempts to delete the managed attribute name on an instance of the owner class; the latter is the general instance destructor method, run when an instance of any kind of class is about to be garbage-collected. Descriptor `__delete__` is more closely related to the `__delattr__` generic attribute deletion method we'll study later in this chapter. See [Chapter 30](#) for more on operator-overloading methods like `__del__`.

A First Example

To see how this all comes together in more realistic code, let's get started with the same first example we wrote for properties. [Example 38-4](#) defines a descriptor that intercepts access to an attribute named `name` in its clients. Its methods use their instance argument to access state information in the subject instance, where the name string is actually stored.

Example 38-4. desc-person.py

```
class Name:
    'name descriptor docs'

    def __get__(self, instance, owner):
        print('fetch...')
        return instance._name

    def __set__(self, instance, value):
        print('change...')
        instance._name = value

    def __delete__(self, instance):
        print('remove...')
        del instance._name

class Person:
    def __init__(self, name):
        self._name = name

    name = Name()                                     # Assign descriptor to attr

sue = Person('Sue Jones')                           # sue has a managed attribute
```

```

print(sue.name)                # Runs Name.__get__
sue.name = 'Susan Jones'      # Runs Name.__set__
print(sue.name)
del sue.name                   # Runs Name.__delete__

print('-'*20)
bob = Person('Bob Smith')      # bob inherits descriptor too
print(bob.name)
print(Name.__doc__)            # Or help(Name)

```

Notice in this code how we assign an instance of our descriptor class to a *class attribute* in the client class; because of this, it is inherited by all instances of the class, just like a class's methods. Really, we *must* assign the descriptor to a class attribute like this—it won't work if assigned to a self instance attribute instead. When the descriptor's `__get__` method is run, it is passed three objects to define its context:

- `self` is the `Name` class instance.
- `instance` is the `Person` class instance.
- `owner` is the `Person` class.

When this code is run, the descriptor's methods intercept accesses to the attribute, much like the property version. In fact, the output is the same again:

```

$ python3 desc-person.py
fetch...
Sue Jones
change...
fetch...
Susan Jones
remove...
-----
fetch...
Bob Smith
name descriptor docs

```

Also like in the property example, our descriptor class instance is a class attribute and thus is *inherited* by all instances of the client class and any subclasses. If we change the `Person` class in our example to the following, for instance, the output of our script is the same:

```

...
class Super:
    def __init__(self, name):
        self._name = name

    name = Name()

class Person(Super):
    # Descriptors are inherited (class attrs)
    pass
...

```

Also, note that when a descriptor class is not useful outside the client class, it's perfectly reasonable to embed the descriptor's definition inside its client syntactically. Here's what our example looks like if we use a *nested class*:

```

class Person:
    def __init__(self, name):
        self._name = name

    class Name:                                # Using a nested class
        'name descriptor docs'

        def __get__(self, instance, owner):
            ...same...

        def __set__(self, instance, value):
            ...same...

        def __delete__(self, instance):
            ...same...

    name = Name()

```

When coded this way, `Name` becomes a local variable in the scope of the `Person` class statement, such that it won't clash with any names outside the class. This version works the same as the original—we've simply moved the descriptor class definition into the client class's scope—but the last line of the testing code must change to fetch the docstring from its new location (per unlisted file *desc-person-nested.py* in the example's package):

```

...
print(Person.Name.__doc__)    # Differs: not Name.__doc__ outside class

```

Computed Attributes

As was the case when using properties, our first descriptor example of the prior section didn't do much—it simply printed trace messages for attribute accesses as a demo. In practice, descriptors can also be used to compute attribute values each time they are fetched. [Example 38-5](#) illustrates—it's a rehash of the same example we coded for properties but uses a descriptor to automatically square an attribute's value each time it is fetched.

Example 38-5. desc-computed.py

```

class DescSquare:
    def __init__(self, start):                # Each desc has own state
        self.value = start

    def __get__(self, instance, owner):       # On attr fetch
        return self.value ** 2

    def __set__(self, instance, value):       # On attr assign
        self.value = value                   # No delete or docs

class Client1:
    X = DescSquare(3)                        # Assign descriptor instance to class attr

class Client2:
    X = DescSquare(32)                       # Another instance in another client class
                                           # Could also code two instances in same class

c1 = Client1()
c2 = Client2()

print(c1.X)                                # 3 ** 2
c1.X = 4

```

```

print(c1.X)           # 4 ** 2
print(c2.X)           # 32 ** 2 (1024)

```

When run, the output of this example is the same as that of the original property-based version, but here a descriptor class object is intercepting the attribute accesses instead of a property:

```

$ python3 desc-computed.py
9
16
1024

```

Using State Information in Descriptors

If you closely study the two descriptor examples we've written so far, you might notice that they get their information from different places—the first (the *name* attribute example) uses data stored on the client *instance*, and the second (the attribute squaring example) uses data attached to the *descriptor* object itself (a.k.a. *self*). In fact, descriptors can use *both* instance state and descriptor state, or any combination thereof:

- *Descriptor state* is used to manage either data internal to the workings of the descriptor or data that spans all instances. It can vary per attribute appearance (often per client class).
- *Instance state* records information related to and possibly created by the client class. It can vary per client-class instance (that is, per application object).

In other words, descriptor state is per-descriptor data, and instance state is per-client-instance data. As usual in OOP, you must choose state carefully. For example, you would not normally use *descriptor* state to record employee names since each client instance requires its own value—if stored in the descriptor, each client class instance will effectively share the same single copy. On the other hand, you would not usually use *instance* state to record data pertaining to descriptor implementation internals—if stored in each instance, there would be multiple varying copies.

Descriptor methods may use either state form, but descriptor state sometimes makes it unnecessary to use special naming conventions to avoid name collisions in the instance for data that is not instance specific. For example, the descriptor in [Example 38-6](#) attaches information to its own instance, so it doesn't clash with that on the client class's instance—but also shares that information between two client instances.

Example 38-6. *desc-state-desc.py*

```

class DescState:                                # Use descriptor state
    def __init__(self, value):
        self.value = value

    def __get__(self, instance, owner):          # On attr fetch
        print('DescState get')
        return self.value * 10

    def __set__(self, instance, value):          # On attr assign
        print('DescState set')
        self.value = value

# Client class
class CalcAttrs:
    X = DescState(2)                            # Descriptor class attr
    Y = 3                                         # Class attr

```

```

def __init__(self):
    self.Z = 4                                # Instance attr

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z)                   # X is computed, others are not
obj.X = 5                                    # X assignment is intercepted
CalcAttrs.Y = 6                              # Y reassigned in class
obj.Z = 7                                    # Z assigned in instance
print(obj.X, obj.Y, obj.Z)

obj2 = CalcAttrs()                           # But X uses shared data, like Y!
print(obj2.X, obj2.Y, obj2.Z)

```

This code's internal value information lives only in the *descriptor*, so there won't be a collision if the same name is used in the client's instance. Notice that only the descriptor attribute is managed here—get and set accesses to *X* are intercepted, but accesses to *Y* and *Z* are not (*Y* is attached to the client class and *Z* to the instance). When this code is run, *X* is computed when fetched, but its value is also the same for all client instances because it uses descriptor-level state:

```

$ python3 desc-state-desc.py
DescState get
20 3 4
DescState set
DescState get
50 6 7
DescState get
50 6 4

```

It's also feasible for a descriptor to store or use an attribute attached to the client class's *instance* instead of itself. Crucially, unlike data stored in the descriptor itself, this allows for data that can vary per client class instance. The descriptor in [Example 38-7](#) assumes the instance has an attribute `_X` attached by the client class and uses it to compute the value of the attribute it represents.

Example 38-7. *desc-state-inst.py*

```

class InstState:
    def __get__(self, instance, owner):      # Using instance state
        print('InstState get')              # Assume set by client class
        return instance._X * 10

    def __set__(self, instance, value):
        print('InstState set')
        instance._X = value

# Client class
class CalcAttrs:
    X = InstState()                          # Descriptor class attr
    Y = 3                                    # Class attr

    def __init__(self):
        self._X = 2                         # Instance attr
        self.Z = 4                          # Instance attr

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z)                 # X is computed, others are not
obj.X = 5                                  # X assignment is intercepted
CalcAttrs.Y = 6                            # Y reassigned in class
obj.Z = 7                                  # Z assigned in instance
print(obj.X, obj.Y, obj.Z)

```

```
obj2 = CalcAttrs()
print(obj2.X, obj2.Y, obj2.Z) # But X differs now, like Z!
```

Here, `X` is assigned to a descriptor as before that manages accesses. The new descriptor here, though, has no information itself, but it uses an attribute assumed to exist in the instance—that attribute is named `_X`, to avoid collisions with the name of the descriptor itself. When this version is run, the results are similar, but the value of the descriptor attribute can vary per client instance due to the differing state policy:

```
$ python3 desc-state-inst.py
InstState get
20 3 4
InstState set
InstState get
50 6 7
InstState get
20 6 4
```

Both descriptor and instance state have roles. In fact, this is a general advantage that descriptors have over properties—because they have state of their own, they can easily retain data internally without adding it to the namespace of the client instance object. As a summary, the following uses *both* state sources—its `self.data` retains per-attribute information, while its `instance.data` can vary per client instance:

```
>>> class DescBoth:
    def __init__(self, data):
        self.data = data
    def __get__(self, instance, owner):
        return f'{self.data}, {instance.data}'
    def __set__(self, instance, value):
        instance.data = value

>>> class Client:
    def __init__(self, data):
        self.data = data
    managed = DescBoth('hack')

>>> I = Client('code')
>>> I.managed # Show both data sources
'hack, code'
>>> I.managed = 'HACK' # Change instance data
>>> I.managed
'hack, HACK'
```

We’ll revisit the implications of this choice in a case study later in this chapter. Before we move on, recall from [Chapter 32](#)’s coverage of *slots* that we can access “virtual” attributes like properties and descriptors with tools like `dir` and `getattr`, even though they don’t exist in the instance’s namespace dictionary. Whether you *should* access these this way probably varies per program—properties and descriptors may run arbitrary computation and may be less obviously instance “data” than slots:

```
>>> I.__dict__
{'data': 'HACK'}
>>> [x for x in dir(I) if not x.startswith('__')]
['data', 'managed']

>>> getattr(I, 'data')
'HACK'
>>> getattr(I, 'managed')
```

```
'hack, HACK'

>>> for attr in (x for x in dir(I) if not x.startswith('__')):
    print(f'{attr} => {getattr(I, attr)}')

data => HACK
managed => hack, HACK
```

The more generic `__getattr__` and `__getattribute__` tools we'll explore soon are not designed to support this functionality: because they have no class-level attributes, their “virtual” attribute names do not appear in `dir` results (per [Chapter 31](#), a `__dir__` can provide a `dir` result, but it's optional and uncommon). In exchange, they are also not limited to specific attribute names coded as properties or descriptors—tools that share even more than this behavior, as the next section explains.

How Properties and Descriptors Relate

As mentioned earlier, properties and descriptors are strongly related—the property built-in is just a convenient way to create a descriptor. Now that you know how both work, you should also be able to see that it's possible to simulate the property built-in with a descriptor class, as demoed by [Example 38-8](#).

Example 38-8. `prop-desc-equiv.py`

```
class Property:
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc
        # Save unbound methods
        # or other callables

    def __get__(self, instance, instancetype=None):
        if instance is None:
            return self
        if self.fget is None:
            raise AttributeError("can't get attribute")
        return self.fget(instance)
        # Pass instance to self
        # in property accessors

    def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(instance, value)

    def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(instance)

class Person:
    def getName(self):
        print('getName...')
    def setName(self, value):
        print('setName...')
    name = Property(getName, setName)
    # Use like property()

x = Person()
x.name
```



```
x.name = 'Pat'
del x.name
```

This `Property` class catches attribute accesses with the descriptor protocol and routes requests to functions or methods passed in and saved in descriptor state when the class's instance is created. Attribute fetches, for example, are routed from the `Person` class, to the `Property` class's `__get__` method, and back to the `Person` class's `getName`. With descriptors, this “just works”:

```
$ python3 prop-desc-equiv.py
getName...
setName...
AttributeError: can't delete attribute
```

Note that this descriptor class equivalent only handles basic property usage, though; to use `@decorator syntax` to also specify set and delete operations, we'd have to extend our `Property` class with `setter` and `deleter` methods, which would save the decorated accessor function and return the property object (`self` should suffice). Since the `property` built-in already does this, we'll omit a formal coding of this extension here.

Descriptors and slots and more

You can also probably now, at least in part, imagine how descriptors are used to implement Python's *slots* extension: instance attribute dictionaries are avoided by creating class-level descriptors that intercept slot name access and map those names to sequential storage space in the instance. Unlike the explicit property call, though, much of the magic behind slots is orchestrated at class creation time both automatically and implicitly when a `__slots__` attribute is present in a class.

See [Chapter 32](#) for more on slots—and why they're not recommended except in pathological use cases. Descriptors are also used for other class tools, but we'll omit further internal details here; see Python's manuals and its open source code for more details.



Descriptor cliff-hangers: In [Chapter 39](#), we'll also make use of descriptors to implement function *decorators* that apply to both functions and methods. As you'll see there, because descriptors receive both descriptor and subject class instances they work well in this role, though nested functions are often a conceptually simpler solution. In addition, [Chapter 39](#) deploys descriptors as one way to intercept *built-in operation* method fetches, and [Chapter 40](#) formalizes data descriptors' precedence in the full *inheritance* model noted earlier: with a `__set__`, descriptors override other names and are thus fairly binding—they cannot be hidden by names in instance dictionaries.

`__getattr__` and `__getattribute__`

So far, we've studied properties and descriptors—tools for managing specific attributes. The `__getattr__` and `__getattribute__` operator-overloading methods provide still other ways to intercept attribute fetches for class instances. Like properties and descriptors, they allow us to insert code to be run automatically when attributes are accessed. As shown here, though, these two methods can also be used in more general ways. Because they intercept arbitrary names, they can apply in broader roles, but may also incur extra calls in some contexts, and are too dynamic to register in `dir` results without help.

This form of attribute-fetch interception comes in two flavors, coded with two different methods:

- `__getattr__` is run for *undefined* attributes—because it is run only for attributes not stored on an instance or inherited from one of its classes, its use is straightforward.
- `__getattribute__` is run for *every* attribute—because it is all-inclusive, you must be cautious when using this method to avoid recursive loops by passing attribute accesses to a superclass.

We met the first of these in [Chapter 30](#). These two methods are representatives of a set of attribute interception methods that also includes `__setattr__` and `__delattr__`. Because these methods have similar roles, though, we will generally treat them all as a single topic here.

Unlike properties and descriptors, these methods are usually considered part of Python’s *operator-overloading* protocol—specially named methods of a class, inherited by subclasses, and run automatically when instances are used in the associated built-in operation (here, attribute fetch). Like all normal methods of a class, they each receive a first `self` argument when called, giving access to both instance state information and other methods of their hosting class.

The `__getattr__` and `__getattribute__` methods are also more *generic* than properties and descriptors—they can be used to intercept access to any (or even all) instance attribute fetches, not just a single specific name. Because of this, these two methods are well suited to general *delegation* coding patterns—they can implement wrapper (a.k.a. *proxy*) objects that manage all attribute accesses for an embedded object. By contrast, we must define one property or descriptor for every attribute we wish to intercept. As covered ahead, this delegation role is limited somewhat for built-in operations but still applies to all named methods in a wrapped object’s interface.

Finally, these two methods are more *narrowly focused* than the alternatives we considered earlier: they intercept attribute fetches only, not assignments. To also catch attribute changes by assignment, we must code a `__setattr__` method—an operator-overloading method run for every attribute assignment, which must take care to avoid recursive loops by routing attribute assignments through the instance namespace dictionary or a superclass method. Although less common, we can also code a `__delattr__` overloading method (which must avoid looping in the same way) to intercept attribute deletions. By contrast, properties and descriptors catch get, set, and delete operations by design.

`__getattr__` and `__setattr__` were introduced in [Chapters 30](#) and [32](#), and `__getattribute__` was mentioned briefly in [Chapter 32](#). Here, we’ll expand on their usage and study their roles in larger contexts.

The Basics

In short, if a class defines or inherits the following methods, they will be run automatically when an instance is used in the operation described by the comments to the right:

```
def __getattr__(self, name):           # On undefined attribute fetch [obj.name]
def __getattribute__(self, name):      # On all attribute fetch [obj.name]
def __setattr__(self, name, value):    # On all attribute assignment [obj.name=value]
def __delattr__(self, name):          # On all attribute deletion [del obj.name]
```

In these, `self` is the subject instance object as usual, `name` is the string name of the attribute being accessed, and `value` is the object being assigned to the attribute. The two get methods normally return an attribute’s value, and the other two return nothing (`None`). All can raise exceptions to signal prohibited access.

For example, to catch every attribute fetch, we can use either of the first two previous methods, and to catch every attribute assignment we can use the third. The following uses `__getattr__` for fetches:

```
class Catcher:
    def __getattr__(self, name):
        print('Get:', name)
    def __setattr__(self, name, value):
        print('Set:', name, value)

X = Catcher()
X.job                # Prints "Get: job"
X.pay                # Prints "Get: pay"
X.pay = 'bread'      # Prints "Set: pay bread"
```

Using `__getattribute__` works exactly the same in this specific case but has subtle looping potential which we'll take up in the next section:

```
class Catcher:
    def __getattribute__(self, name):
        print('Get:', name)
    ...rest unchanged...

# On all attribute fetches
# Works same as getattr here
# But prone to loops in general
```

Such a coding structure can be used to implement the *delegation* design pattern we met earlier in [Chapter 31](#). Because all attributes are routed to interception methods generically, we can validate and pass them along to embedded, managed objects. As a refresher, the following class, borrowed from [Chapter 31](#), traces *every* attribute fetch made to another object passed to the wrapper (proxy) class:

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object
    def __getattr__(self, attrname):
        print('Trace:', attrname)
        return getattr(self.wrapped, attrname)

X = Wrapper([1, 2, 3])
X.append(4)
print(X.wrapped)

# Save object
# Trace fetch
# Delegate fetch

# Prints "Trace: append"
# Prints "[1, 2, 3, 4]"
```

There is no such analog for properties and descriptors, short of coding accessors for *every* attribute present in *every* wrapped object. On the other hand, when such generality is not required, generic accessor methods may incur additional calls for assignments in some contexts—a trade-off described in [Chapter 30](#) and mentioned in the context of the case study example we'll explore at the end of this chapter.

Avoiding loops in attribute interception methods

These methods are generally straightforward to use. Their most complex aspect is the potential for *looping* (a.k.a. recursing). Because `__getattr__` is called for undefined attributes only, it can freely fetch other attributes within its own code. However, because `__getattribute__` and `__setattr__` are run for *all* attributes, their code must be careful when accessing other attributes to avoid calling themselves again and triggering a recursive loop.

For example, another attribute fetch run inside a `__getattribute__` method's code like the following will trigger `__getattribute__` again—and the code will usually loop until memory is exhausted:

```
def __getattribute__(self, name):
    x = self.other

# LOOPS!
```

Technically, this method is even more loop-prone than this may imply—a `self` attribute reference run *anywhere* in a class that defines this method will trigger `__getattribute__` and also has the potential to loop, depending on the class's logic. This is normally desired behavior—intercepting every attribute fetch is this method's purpose, after all—but you should be aware that this method catches *all* attribute fetches wherever they are coded. When coded within `__getattribute__` itself, this almost always causes a loop.

To avoid this loop, route the fetch through a higher superclass instead to skip this level's version—because the object class is always a superclass to every class, it serves well in this role:

```
def __getattribute__(self, name):
    x = object.__getattribute__(self, 'other')    # Force higher to avoid me
```

For `__setattr__`, the situation is similar, as summarized in [Chapter 30](#)—assigning *any* attribute inside this method triggers `__setattr__` again and may create a similar loop:

```
def __setattr__(self, name, value):
    self.other = value                          # Recurs (and might LOOP!)
```

Here too, `self` attribute assignments *anywhere* in a class defining this method trigger `__setattr__` as well, though the potential for looping is much stronger when they show up in `__setattr__` itself. To work around this problem, you can assign the attribute as a key in the instance's `__dict__` namespace dictionary instead. This avoids direct attribute assignment:

```
def __setattr__(self, name, value):
    self.__dict__['other'] = value              # Use attr dict to avoid me
```

Alternatively, `__setattr__` can also pass its own attribute assignments to a higher superclass to avoid looping, just like `__getattribute__`. In fact, this scheme is sometimes preferred when wrapped classes use *slots*, *properties*, or other “virtual” attributes that live on classes instead of instances—and in the case of slots, may preclude `__dict__`:

```
def __setattr__(self, name, value):
    object.__setattr__(self, 'other', value)    # Force higher to avoid me
```

This book's `__setattr__` examples often use `__dict__` for smaller demos anyhow, just because their parameters are known. By contrast, though, we *cannot* use the `__dict__` trick to avoid loops in `__getattribute__`:

```
def __getattribute__(self, name):
    x = self.__dict__['other']                  # Loops!
```

If this is coded, fetching the `__dict__` attribute itself triggers `__getattribute__` again—causing a recursive loop and an immediate fail. Strange but true!

The `__delattr__` method is less commonly used in practice, but when it is, it is called for *every* attribute deletion, just as `__setattr__` is called for every attribute assignment. When using this method, you must avoid loops when deleting attributes by the same techniques: namespace dictionaries operations or superclass method calls.

A First Example

Generic attribute management is not nearly as complicated as the prior section may have implied. To see how to put these ideas to work, [Example 38-9](#) is the same first example we used for properties and descriptors in action again, this time implemented with attribute operator-overloading methods. Because these methods are so generic, we test attribute names here to know when a managed attribute is being accessed; others are allowed to pass normally.

Example 38-9. *getattr-person.py*

```
class Person:
    def __init__(self, name):           # On [Person()]
        self._name = name              # Triggers __setattr__!

    def __getattr__(self, attr):        # On [obj.undefined]
        print('get: ' + attr)
        if attr == 'name':             # Intercept name: not stored
            return self._name          # Does not loop: real attr
        else:                           # Others are errors
            raise AttributeError(attr)

    def __setattr__(self, attr, value):  # On [obj.any = value]
        print('set: ' + attr)
        if attr == 'name':
            attr = '_name'              # Set internal name
        self.__dict__[attr] = value     # Avoid looping here

    def __delattr__(self, attr):         # On [del obj.any]
        print('del: ' + attr)
        if attr == 'name':
            attr = '_name'              # Avoid looping here too
        del self.__dict__[attr]         # but much less common

sue = Person('Sue Jones')              # sue has a managed attribute
print(sue.name)                        # Runs __getattr__
sue.name = 'Susan Jones'               # Runs __setattr__
print(sue.name)
del sue.name                           # Runs __delattr__

print('-'*20)
bob = Person('Bob Smith')              # bob's attrs work like sue's
print(bob.name)
#print(Person.name.__doc__)             # No direct equivalent here!
```

When this code is run, the same sort of output is produced, but this time it reflects our generic attribute-interception methods responding to Python's normal operator-overloading mechanism:

```
$ python3 getattr-person.py
set: _name
get: name
Sue Jones
set: name
get: name
Susan Jones
del: name
-----
set: _name
get: name
Bob Smith
```

Notice how the attribute assignment in the `__init__` constructor triggers `__setattr__` too—this method catches *every* instance-attribute assignment, even those anywhere within the class itself, and those to underlying attributes like `_name`. Also note that, unlike with properties and descriptors, there's no direct notion of specifying *documentation* for our attribute here; managed attributes exist within the code of our interception methods, not as distinct objects.

Using `__getattribute__`

To achieve exactly the same results with `__getattribute__`, replace `__getattr__` in [Example 38-9](#) with the differing code in [Example 38-10](#). Because it catches *all* attribute fetches, this version must be careful to avoid looping by passing new fetches to a superclass, and it can't generally assume unknown names are errors.

Example 38-10. `getattribute-person.py` (differing part)

```
# Replace just __getattr__ with this

def __getattribute__(self, attr):                # On [obj.any]
    print('get: ' + attr)
    if attr == 'name':                          # Intercept all names
        attr = '_name'                        # Map to internal name
    return object.__getattribute__(self, attr)   # Avoid looping here
```

When run with this change, the output is similar, but we get an extra `__getattribute__` call for the fetch of `__dict__` in `__setattr__` (the first time originating in `__init__`):

```
$ python3 getattribute-person.py
set: _name
get: __dict__
get: name
Sue Jones
set: name
get: __dict__
get: name
Susan Jones
del: name
get: __dict__
-----
set: _name
get: __dict__
get: name
Bob Smith
```

This example is equivalent to that coded for properties and descriptors, but it's a bit artificial, and it doesn't really highlight these tools' assets. Because they are generic, `__getattr__` and `__getattribute__` are probably more commonly used in delegation-base code (as sketched earlier), where attribute access is validated and routed to an embedded object. Where just a *single* attribute must be managed, properties and descriptors might do as well or better, and avoid extra calls for unmanaged attributes.

Computed Attributes

As before, our prior example doesn't really do anything but trace attribute fetches; it's not much more work to compute an attribute's value when fetched. As for properties and descriptors, [Example 38-11](#) creates a virtual attribute `X` that runs a calculation when fetched.

Example 38-11. *getattr-computed.py*

```
class AttrSquare:
    def __init__(self, start):
        self.value = start                # Triggers __setattr__!

    def __getattr__(self, attr):
        if attr == 'X':
            return self.value ** 2        # value is not undefined
        else:
            raise AttributeError(attr)

    def __setattr__(self, attr, value):    # On all attr assignments
        if attr == 'X':
            attr = 'value'
            self.__dict__[attr] = value

A = AttrSquare(3)                        # 2 instances of class with overloading
B = AttrSquare(32)                       # Each has different state information

print(A.X)                               # 3 ** 2
A.X = 4
print(A.X)                               # 4 ** 2
print(B.X)                               # 32 ** 2 (1024)
```

Running this code results in the same output that we got earlier when using properties and descriptors, but this script's mechanics are based on generic attribute interception methods:

```
$ python3 getattr-computed.py
9
16
1024
```

Using `__getattribute__`

As before, we can achieve the same effect with `__getattribute__` instead of `__getattr__`. **Example 38-12** replaces the fetch method with a `__getattribute__` and changes the `__setattr__` assignment method to avoid looping by using direct object superclass method calls instead of `__dict__` keys.

Example 38-12. *getattribute-computed.py*

```
class AttrSquare:
    def __init__(self, start):
        self.value = start                # Triggers __setattr__!

    def __getattribute__(self, attr):
        if attr == 'X':
            return self.value ** 2        # Triggers __getattribute__ again!
        else:
            return object.__getattribute__(self, attr)

    def __setattr__(self, attr, value):    # On all attr assignments
        if attr == 'X':
            attr = 'value'
            object.__setattr__(self, attr, value)
```

...self-test code same as Example 38-11...

When this version is run, the results are the same again so we won't relist them here. Notice, though, the implicit and subtle routing going on inside this class's methods:

- `self.value=start` inside the constructor triggers `__setattr__`.
- `self.value` inside `__getattr__` triggers `__getattr__` again.

In fact, `__getattr__` is run *twice* each time we fetch attribute X. This doesn't happen in the `__getattribute__` version because the `value` attribute is not undefined (and hence skips the method). If you care about speed and want to avoid this, change `__getattr__` to use the superclass to fetch `value` as well:

```
def __getattribute__(self, attr):
    if attr == 'X':
        return object.__getattribute__(self, 'value') ** 2
```

Of course, this still incurs a call to the superclass method but not an additional recursive call before we get there. If that's confusing, add `print` calls to these methods to trace how and when they run.

`__getattr__` and `__getattribute__` Compared

To summarize the coding differences between `__getattr__` and `__getattribute__`, [Example 38-13](#) uses both to implement three attributes—`attr1` is a class attribute, `attr2` is an instance attribute, and `attr3` is a virtual managed attribute computed when fetched.

Example 38-13. `getattr-v-getattribute.py`

```
class GetAttr:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattr__(self, attr):
        print('get:', attr)
        if attr == 'attr3':
            return 3
        else:
            raise AttributeError(attr)

X = GetAttr()
print(X.attr1)
print(X.attr2)
print(X.attr3)
print('-'*20)

class GetAttribute:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattribute__(self, attr):
        print('get:', attr)
        if attr == 'attr3':
            return 3
        else:
            return object.__getattribute__(self, attr)

X = GetAttribute()
print(X.attr1)
```



```
print(X.attr2)
print(X.attr3)
```

When run, the `__getattr__` version intercepts only `attr3` accesses because it is undefined. The `__getattribute__` version, on the other hand, intercepts all attribute fetches and must route those it does not manage to the superclass fetcher to avoid loops:

```
$ python3 getattr-v-getattribute.py
1
2
get: attr3
3
-----
get: attr1
1
get: attr2
2
get: attr3
3
```

Although `__getattribute__` can catch more attribute fetches than `__getattr__`, in practice they are often just variations on a theme—if attributes are not physically stored, the two have the same effect.

Management Techniques Compared

To summarize the coding differences in all four attribute-management schemes we’ve just explored, let’s quickly step through a somewhat more comprehensive computed-attribute example using each technique. The first version, [Example 38-14](#), uses *properties* to intercept and calculate attributes named `square` and `cube`. Notice how their base values are stored in names that begin with an underscore so they don’t clash with the names of the properties themselves.

Example 38-14. all_four_props.py

"Two dynamically computed attributes with properties"

```
class Powers:
    def __init__(self, square, cube):
        self._square = square           # _square is the base value
        self._cube   = cube             # square is the property name

    def getSquare(self):
        return self._square ** 2

    def setSquare(self, value):
        self._square = value

    square = property(getSquare, setSquare)  # Or @property decorator

    def getCube(self):
        return self._cube ** 3

    cube = property(getCube)               # Likewise
```

To do the same with *descriptors*, [Example 38-15](#) defines the attributes with complete classes. Note that these descriptors store base values as instance state, so they must use leading underscores again so as not to clash with the names of descriptors; as called out by the final example of this chapter, we could avoid this renaming requirement by storing base values as descriptor state instead, but that doesn’t as directly address data that must vary per client-class instance.

Example 38-15. *all_four_desc.py*

"Same, but with descriptors (per-instance state)"

```
class DescSquare:
    def __get__(self, instance, owner):
        return instance._square ** 2
    def __set__(self, instance, value):
        instance._square = value

class DescCube:
    def __get__(self, instance, owner):
        return instance._cube ** 3

class Powers:
    square = DescSquare()
    cube = DescCube()
    def __init__(self, square, cube):
        self._square = square          # "self.square = square" works too,
        self._cube = cube              # because it triggers desc __set__!
```

To achieve the same result with `__getattr__` fetch interception, [Example 38-16](#) again stores base values with underscore-prefixed names so that accesses to managed names are undefined and thus invoke its method; it also needs to code a `__setattr__` to intercept assignments and take care to avoid its potential for looping.

Example 38-16. *all_four_getattr.py*

"Same, but with generic `__getattr__` undefined-attribute interception"

```
class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube

    def __getattr__(self, name):
        if name == 'square':
            return self._square ** 2
        elif name == 'cube':
            return self._cube ** 3
        else:
            raise TypeError('unknown attr:' + name)

    def __setattr__(self, name, value):
        if name == 'square':
            self.__dict__['_square'] = value          # Or use object
        else:
            self.__dict__[name] = value
```

The final option in [Example 38-17](#), coding with `__getattribute__`, is similar to the prior version. Because it catches every attribute now, though, it must also route base value fetches to a superclass to avoid looping or extra calls—fetching `self._square` directly works too, but runs a second `__getattribute__` call.

Example 38-17. *all_four_getattribute.py*

"Same, but with generic `__getattribute__` all-attribute interception"

```
class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube

    def __getattribute__(self, name):
        if name == 'square':
            return object.__getattribute__(self, '_square') ** 2
        elif name == 'cube':
            return object.__getattribute__(self, '_cube') ** 3
        else:
            return object.__getattribute__(self, name)

    def __setattr__(self, name, value):
        if name == 'square':
            object.__setattr__(self, '_square', value)    # Or use __dict__
        else:
            object.__setattr__(self, name, value)
```

To test, the following REPL session loops through a list of all four modules' name strings and imports and fetches classes along the way. Each technique takes a different form in code, but all four produce the same result when run:

```
>>> from importlib import import_module
>>> mods = [f'all_four_{M}' for M in ('props', 'desc', 'getattr', 'getattribute')]
>>> for modname in mods:
    module = import_module(modname)    # Import by name string
    X = module.Powers(3, 4)            # This module's class (print to see)
    print(X.square)                    # 3 ** 2 = 9
    print(X.cube)                      # 4 ** 3 = 64
    X.square = 5
    print(X.square)                    # 5 ** 2 = 25

9
64
25
...repeated four times...
```

For more on how these alternatives compare, and other coding options, stay tuned for a more realistic application of them in the attribute-validation example ahead. First, though, we need to take a short side trip to study a pitfall associated with two of these tools—the generic attribute interceptors.

Intercepting Built-in Operation Attributes

If you've been reading this book linearly, some of this section is elaboration on earlier notes, especially the sidebar **"Delegating Built-ins—or Not" on page 698**. When `__getattr__` and `__getattribute__` were introduced here, it was stated that they intercept undefined- and all-attribute fetches, respectively, which makes them ideal for delegation-based coding patterns.

While this is true for both *normally named* and *explicitly fetched* attributes, their behavior needs some additional clarification. Specifically, the implicit method-name fetches of *built-in operations* will never automatically be routed to either of these two attribute-interceptor methods. This means that operator-overloading method calls cannot be delegated to wrapped objects unless wrapper classes somehow redefine these methods themselves.

For example, attribute fetches for the `__str__`, `__add__`, and `__getitem__` methods run *implicitly* by printing, `+` expressions, and indexing, respectively, are not routed to either `__getattr__` or `__getattribute__`. Instead, such methods are looked up in classes, and skip the instance and its attribute interceptors. Hence, there is no direct way to generically catch and delegate built-in operations like these.

This was a Python 3.X bifurcation, whose purported rationale involved metaclasses and optimization of built-in operations. Whatever its basis, all attributes—both `__X__` and other—are still dispatched through the instance’s interceptor methods when accessed *explicitly* by name, so this qualifies as a glaring inconsistency: `X.__add__` runs `__getattr__`, but `X+Y`, which uses `X.__add__`, does not. The net effect complicates delegation-based code.

The good news is that wrapper classes can work around this constraint by redefining operator-overloading methods in the wrapper itself, in order to catch and delegate calls. These extra methods can be added either manually, with tools, or by definition in, and inheritance from, common superclasses. It’s more work for delegation classes when operator-overloading methods are part of a wrapped object’s interface, but it’s not a showstopper.

As a demo of the issue, consider the code in [Example 38-18](#), which tests various attribute types and built-in operations on instances of classes containing `__getattr__` and `__getattribute__` methods.

Example 38-18. `getattr-builtins.py`

```
class GetAttr:
    cattr = 88                                # Attrs stored on class and instance
    def __init__(self):                       # These skip getattr, but not getattribute
        self.iattr = 77

    def __len__(self):                        # Redefine for len(): doesn't run getattr
        print('__len__: 66')
        return 66

    def __getattr__(self, attr):              # Provide __str__ if asked, else dummy func
        print('getattr:', attr)              # Never run for __str__: inherited from object
        if attr == '__str__':
            return lambda *args: '[GetAttr str]'
        else:
            return lambda *args: None

class GetAttribute:
    cattr = 88                                # Similar, but catch all attributes
    def __init__(self):                       # Except implicit fetches for built-in ops
        self.iattr = 77

    def __len__(self):                        # Redefine for len(): doesn't run getattribute
        print('__len__: 66')                  # But explicit fetches of inherited __str__ do
        return 66

    def __getattribute__(self, attr):
        print('getattribute:', attr)
        if attr == '__str__':
            return lambda *args: '[GetAttribute str]'
        else:
            return lambda *args: None

for Class in GetAttr, GetAttribute:
    print('\n' + Class.__name__.ljust(50, '='))
```

```

X = Class()

# Defined attributes trigger getattr but not getatttr

X.cattr          # Class attr    (defined - skips getattr)
X.iattr          # Instance attr (defined - skips getattr)
X.other          # Missing attr
len(X)           # __len__ defined explicitly: moot

# Built-in ops do not invoke either getattr or getattribute
# No defaults are inherited for these from object superclass

try:    X[0]          # Tries to invoke __getitem__
except: print('fail []')
try:    X + 99        # Ditto, __add__
except: print('fail +')
try:    X()           # Ditto, __call__
except: print('fail ()')

# But explicit calls invoke both catchers

X.__getitem__(0)
X.__add__(99)
X.__call__()

# The implied object superclass defines a __str__ that precludes getattr
# But the absolute getattribute is not called for implicit fetches either

print(X.__str__())    # __str__: explicit call => only __getattr__ skipped
print(X)              # __str__: implicit via built-in => both skipped

```

This file runs the same set of tests on each of its classes in turn. Match its following output with its tests and comments to see how it works. In short, neither `__getattr__` nor `__getattribute__` are run for any of the operator-overloading names invoked by built-in operations because such names are looked up in classes only:

```

$ python3 getattr-builtins.py

GetAttr=====
getattr: other
__len__: 66
fail []
fail +
fail ()
getattr: __getitem__
getattr: __add__
getattr: __call__
<__main__.GetAttr object at 0x10f76f020>
<__main__.GetAttr object at 0x10f76f020>

GetAttribute=====
getattribute: cattr
getattribute: iattr
getattribute: other
__len__: 66
fail []
fail +
fail ()
getattribute: __getitem__
getattribute: __add__

```

```

getattribute: __call__
getattribute: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x10f74c440>

```

More generally, all *explicit* method-name attribute fetches are always routed to both attribute-interception methods, but none of the *implicit* operator-overloading methods trigger either attribute-interception method when their attributes are fetched by built-in operations. Salient points in this demo worth calling out:

- `__str__` access fails to be caught twice by `__getattr__`: once for the built-in `print`, and once for explicit fetches because a default is inherited from the built-in object implied above every topmost class.
- `__str__` fails to be caught only once by the `__getattribute__` catchall—during the built-in `print` operation. Explicit fetches bypass the inherited `__str__` and run `__getattribute__`.
- `__call__` fails to be caught in both schemes for built-in call expressions, but it is intercepted by both when fetched explicitly; unlike `__str__`, there is no inherited `__call__` default in object to defeat `__getattr__` in explicit fetches. The same goes for the `__add__` of `+` operations.
- `__len__` is handled by both classes because it is an explicitly defined method in the classes themselves—though its name is not routed to either `__getattr__` or `__getattribute__` if we delete the classes’ `__len__` methods because the `len` built-in skips them as usual.

Again, the net effect is that operator-overloading methods implicitly run by built-in operations are never routed through either attribute interception method. Python begins the search for such attributes in *classes* and skips instance lookup mechanisms entirely. Normally, named attributes and explicit fetches start with the instance instead.

For a more realistic example of this phenomenon’s impact on delegation classes, stay tuned for [Chapter 39](#)’s Private decorator—along with its coverage of multiple reusable *workarounds*.

Revisiting Chapter 28’s delegation example

As a coda, you should also now be able to work out why the `Manager` class coded in [Example 28-11](#) of [Chapter 28](#) had to code a `__repr__` to route printing requests to its wrapped object. Just like `__str__` in our demo, `object` provides a default `__repr__`, which would prevent `print` operations from invoking a `__getattr__`. Technically speaking, `object` defines both `__str__` and `__repr__`, but its `__str__` simply calls `__repr__`.

That said, `object`’s defaults are largely a moot point: like all built-in operations, `print` bypasses both `__getattr__` and `__getattribute__`, as it did for `__getattribute__` in our demo. Hence, a `__repr__` is required by *both* the `object` default and the built-in’s behavior.

Again, fixes for delegating built-ins are in [Chapter 39](#) (unless we run out of underscores before that!).

Example: Attribute Validations

To close out this chapter, let’s turn to a more realistic example, coded in all four of our attribute management schemes. The example we will use defines a `CardHolder` object with four attributes, three of which are managed. The managed attributes validate or transform values when fetched or stored. All four versions produce the same results for the same test code, but they implement their attributes in

very different ways. The examples are largely for self-study; although we won't go through their code in detail, they all use concepts we've already explored in this chapter.

Using Properties to Validate

Our first coding in [Example 38-19](#) uses properties to manage three attributes. As usual, we could use simple methods instead of managed attributes, but properties help if we have already been using attributes in existing code. Properties run code automatically on attribute access but are focused on a specific set of attributes; they cannot be used to intercept all attributes generically.

To understand this code, it's crucial to notice that the attribute assignments inside the `__init__` constructor method trigger property setter methods too. When this method assigns to `self.name`, for example, it automatically invokes the `setName` method, which transforms the value and assigns it to an instance attribute called `__name` so it won't clash with the property's name.

This renaming, sometimes called *name mangling*, is important because properties use common instance state and have none of their own. Data is stored in an attribute called `__name`, and the attribute called `name` is always a property, not data. As we saw in [Chapter 31](#), names like `__name` are known as *pseudo-private* attributes and are changed by Python to include the enclosing class's name when stored in the instance's namespace; here, this helps keep the implementation-specific attributes distinct from others, including that of the property that manages them.

In the end, this class manages attributes called `name`, `age`, and `acct`; allows the attribute `addr` to be accessed directly; and provides a read-only attribute called `remain` that is entirely virtual and computed on demand. For comparison purposes, this property-based coding weighs in at 39 lines of code (including blank lines).

Example 38-19. validate_properties.py

```
class CardHolder:
    acctlen = 8                # Class data
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct      # Instance data
        self.name = name      # These trigger prop setters too!
        self.age = age        # __X mangled to have class name
        self.addr = addr      # addr is not managed
                                # remain has no data

    def getName(self):
        return self.__name
    def setName(self, value):
        value = value.lower().replace(' ', '_')
        self.__name = value
    name = property(getName, setName)    # Or @ decorators for both

    def getAge(self):
        return self.__age
    def setAge(self, value):
        if value < 0 or value > 150:
            raise ValueError('invalid age')
        else:
            self.__age = value
    age = property(getAge, setAge)

    def getAcct(self):
```

```

        return self.__acct[:-3] + '***'
    def setAcct(self, value):
        value = value.replace('-', '')
        if len(value) != self.acctlen:
            raise TypeError('invalid acct number')
        else:
            self.__acct = value
    acct = property(getAcct, setAcct)

    def remainGet(self):
        return self.retireage - self.age
    remain = property(remainGet)

```

Testing code

To test our class, run the script in [Example 38-20](#) in a console with the name of the class’s module (sans “py”) as a single command-line argument (you could also import the class in a REPL, but we’re trying to avoid repeating code here). We’ll use this same test script for all four versions of this example so their output will be the same. When it runs, it makes two instances of our managed-attribute class and fetches and changes their various attributes. Operations expected to fail are wrapped in try statements.

Example 38-20. validate_tester.py

```

def loadclass():
    import sys, importlib
    modulename = sys.argv[1]
    module = importlib.import_module(modulename)
    print(f'[Using: {module.CardHolder}]')
    return module.CardHolder

def printholder(who):
    print(who.acct, who.name, who.age, who.remain, who.addr, sep=' / ')

if __name__ == '__main__':
    CardHolder = loadclass()
    bob = CardHolder('1234-5678', 'Bob Smith', 40, '123 main st')
    printholder(bob)
    bob.name = 'Bob Q. Smith'
    bob.age = 50
    bob.acct = '23-45-67-89'
    printholder(bob)

    sue = CardHolder('5678-12-34', 'Sue Jones', 35, '124 main st')
    printholder(sue)
    try:
        sue.age = 200
    except: print('Bad age for Sue')

    try:
        sue.remain = 5
    except: print("Can't set sue.remain")

    try:
        sue.acct = '1234567'
    except: print('Bad acct for Sue')

```

Following is the output of our test script’s code; again, this is the same for the other versions of this example ahead, except for the tested class’s name. Trace through this code to see how the class’s

methods are invoked. Accounts are displayed with some digits hidden, names are converted to a standard format, and time remaining until retirement (hypothetically speaking) is computed when fetched using a class-attribute cutoff:

```
$ python3 validate_tester.py validate_properties
[Using: <class 'validate_properties.CardHolder'>]
12345*** / bob_smith / 40 / 22.5 / 123 main st
23456*** / bob_q._smith / 50 / 12.5 / 123 main st
56781*** / sue_jones / 35 / 27.5 / 124 main st
Bad age for Sue
Can't set sue.remain
Bad acct for Sue
```

Using Descriptors to Validate

Now, let's recode our example using *descriptors* instead of properties. As we've seen, descriptors are very similar to properties in terms of functionality and roles; in fact, properties are basically a focused form of descriptor. Like properties, descriptors are designed to handle specific attributes, not generic attribute access. Unlike properties, descriptors can also have their own state, and so are perhaps a more general scheme.

Option 1: Validating with shared descriptor-instance state (badly!)

To understand the code in [Example 38-21](#), it's again important to notice that the attribute assignments inside the `__init__` constructor method trigger descriptor `__set__` methods. When the constructor method assigns to `self.name`, for example, it automatically invokes the `Name.__set__()` method, which transforms the value and assigns it to a descriptor attribute called `name`.

In the end, this class implements the same attributes as the prior version: it manages attributes called `name`, `age`, and `acct`; allows the attribute `addr` to be accessed directly; and provides a read-only attribute called `remain` that is entirely virtual and computed on demand. Notice how we must catch assignments to the `remain` name in its descriptor and raise an exception; as we learned earlier, if we did not do this, assigning to this attribute of an instance would silently create an instance attribute that hides the class-attribute descriptor.

For comparison purposes, this descriptor-based coding takes 45 lines of code.

Example 38-21. validate_descriptors1.py

```
class CardHolder:                                # Using shared descriptor state
    acctlen = 8                                  # Class data
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                        # Instance data
        self.name = name                       # These trigger __set__ calls too!
        self.age = age                         # __X not needed: in descriptor
        self.addr = addr                      # addr is not managed
                                                # remain has no data

    class Name:
        def __get__(self, instance, owner):    # Class names: CardHolder locals
            return self.name
        def __set__(self, instance, value):
            value = value.lower().replace(' ', '_')
            self.name = value
name = Name()
```

```

class Age:
    def __get__(self, instance, owner):
        return self.age                # Use descriptor data
    def __set__(self, instance, value):
        if value < 0 or value > 150:
            raise ValueError('invalid age')
        else:
            self.age = value
age = Age()

class Acct:
    def __get__(self, instance, owner):
        return self.acct[:-3] + '***'
    def __set__(self, instance, value):
        value = value.replace('-', '')
        if len(value) != instance.acctlen:    # Use instance class data
            raise TypeError('invalid acct number')
        else:
            self.acct = value
acct = Acct()

class Remain:
    def __get__(self, instance, owner):
        return instance.retireage - instance.age    # Triggers Age.__get__
    def __set__(self, instance, value):
        raise TypeError('cannot set remain')        # Else set allowed here
remain = Remain()

```

When run with the prior testing script, all examples in this section produce the same output as shown for properties earlier, except that the name of the class in the first line varies:

```

$ python3 validate_tester.py validate_descriptors1
[Using: <class 'validate_descriptors1.CardHolder'>]
...rest is same output as properties...

```

Option 2: Validating with per-client-instance state (correctly)

Unlike in the prior property-based variant, though, in [Example 38-21](#), the actual `name` value is attached to the *descriptor* object, not the client class instance. Although we could store this value in either instance or descriptor state, the latter avoids the need to mangle names with underscores to avoid collisions. In the `CardHolder` client class, the attribute called `name` is always a descriptor object, not data.

Importantly, the downside of this scheme is that state stored inside a descriptor itself is class-level data that is effectively *shared* by all client-class instances and so cannot vary between them. That is, storing state in the *descriptor* instance instead of the *owner* (client) class instance means that the state will be the same in all owner-class instances. Descriptor state can vary only per attribute appearance.

To see this at work, try printing attributes of the `bob` instance after creating the second instance, `sue`, with the new test script in [Example 38-22](#). The values of `sue`'s managed attributes (`name`, `age`, and `acct`) *overwrite* those of the earlier object `bob`, because both share the same, single descriptor instance attached to their class.

Example 38-22. *validate_tester_plus.py*

```
from validate_tester import loadclass
CardHolder = loadclass()

bob = CardHolder('1234-5678', 'Bob Smith', 40, '123 main st')
print('bob:', bob.name, bob.acct, bob.age, bob.addr)

sue = CardHolder('5678-12-34', 'Sue Jones', 35, '124 main st')
print('sue:', sue.name, sue.acct, sue.age, sue.addr)    # addr differs: client data
print('bob:', bob.name, bob.acct, bob.age, bob.addr)    # name,acct,age overwritten?
```

When this script is run with the descriptor-state `CardHolder` of [Example 38-21](#), the results confirm the suspicion—in terms of managed attributes, bob has morphed into sue!

```
$ python3 validate_tester_plus.py validate_descriptors1
[Using: <class 'validate_descriptors1.CardHolder'>]
bob: bob_smith 12345*** 40 123 main st
sue: sue_jones 56781*** 35 124 main st
bob: sue_jones 56781*** 35 123 main st
```

This isn't an issue for properties because they have no state of their own, and there are valid uses for descriptor state. Such state might be used, for example, to manage descriptor implementation and data that spans all instances, and this example was coded this way on purpose to illustrate the technique. Moreover, the state scope implications of class versus instance attributes should be more or less a given at this point in the book.

However, in this particular use case, attributes of `CardHolder` objects are probably better stored as *per-instance* data instead of descriptor-instance data, perhaps using the same `__X` naming convention as the property-based equivalent to avoid name clashes in the instance—a more important factor this time, as the client is a different class with its own state attributes. [Example 38-23](#) has the required changes; it doesn't change line counts (we're still at 45).

Example 38-23. *validate_descriptors2.py*

```
class CardHolder:                                # Using per-client-instance state
    acctlen = 8                                   # Class data
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                         # Client instance data
        self.name = name                        # These trigger __set__ calls too!
        self.age = age                          # __X needed: in client instance
        self.addr = addr                       # addr is not managed
                                                # remain managed but has no data

    class Name:
        def __get__(self, instance, owner):      # Class names: CardHolder locals
            return instance.__name
        def __set__(self, instance, value):
            value = value.lower().replace(' ', '_')
            instance.__name = value

    name = Name()                                # class.name vs mangled attr

    class Age:
        def __get__(self, instance, owner):
            return instance.__age                # Use *instance* data
        def __set__(self, instance, value):
            if value < 0 or value > 150:
```

```

        raise ValueError('invalid age')
    else:
        instance.__age = value
age = Age()                                # class.age vs mangled attr

class Acct:
    def __get__(self, instance, owner):
        return instance.__acct[:-3] + '***'
    def __set__(self, instance, value):
        value = value.replace('-', '')
        if len(value) != instance.acctlen:    # Use instance class data
            raise TypeError('invalid acct number')
        else:
            instance.__acct = value
acct = Acct()                               # class.acct vs mangled name

class Remain:
    def __get__(self, instance, owner):
        return instance.retireage - instance.age    # Triggers Age.__get__
    def __set__(self, instance, value):
        raise TypeError('cannot set remain')        # Else set allowed here
remain = Remain()

```

This supports per-instance data for the name, age, and acct managed fields as expected (bob remains bob), and other tests work as before:

```
$ python3 validate_tester_plus.py validate_descriptors2
```

```
[Using: <class 'validate_descriptors2.CardHolder'>]
```

```
bob: bob_smith 12345*** 40 123 main st
```

```
sue: sue_jones 56781*** 35 124 main st
```

```
bob: bob_smith 12345*** 40 123 main st
```

```
$ python3 validate_tester.py validate_descriptors2
```

```
...same output as properties, except class name...
```

One small caveat here: as coded, this version doesn't support *through-class* descriptor access because such access passes a None to the instance argument (also notice the attribute `__X` name mangling to `_Name__name` in the error message when the fetch attempt is made):

```

>>> from validate_descriptors1 import CardHolder
>>> pat = CardHolder('1234-5678', 'Pat Smith', 40, '123 main st')
>>> pat.name
'pat_smith'
>>> CardHolder.name
'pat_smith'

>>> from validate_descriptors2 import CardHolder
>>> pat = CardHolder('1234-5678', 'Pat Smith', 40, '123 main st')
>>> pat.name
'pat_smith'
>>> CardHolder.name
AttributeError: 'NoneType' object has no attribute '_Name__name'

```

We could detect this with a minor amount of additional code to trigger the error more explicitly, but there's probably no point—because this version stores data in the *client instance*, there's no meaning to its descriptors unless they're accompanied by a client instance (much like a normal nonbound instance method). In fact, that's really the entire point of this version's change!

Because they are classes, descriptors are a useful and powerful tool, but they present choices that can deeply impact a program's behavior. As always in OOP, choose your state retention policies carefully.

Using `__getattr__` to Validate

As we've seen, the `__getattr__` method intercepts all undefined attributes, so it can be more generic than using properties or descriptors. For our example, we simply test the attribute name to know when a managed attribute is being fetched; others are stored physically on the instance and so never reach `__getattr__`. Although this approach is more general than using properties or descriptors, extra work may be required to imitate the specific attribute focus of other tools. We need to check names at run-time—a multiple-choice that's a prime role for the `match` statement—and we must code a `__setattr__` in order to intercept and validate attribute assignments.

Example 38-24 hosts the `__getattr__` version of our validations code. In the end, this class, like the prior two, manages attributes called `name`, `age`, and `acct`; allows the attribute `addr` to be accessed directly; and provides a read-only attribute called `remain` that is entirely virtual and is computed on demand.

As for the property and descriptor versions of this example, it's critical to notice that the attribute assignments inside the `__init__` constructor method trigger the class's `__setattr__` method too. When this method assigns to `self.name`, for example, it automatically invokes the `__setattr__` method, which transforms the value and assigns it to an instance attribute called `name`. By storing `name` on the instance, it ensures that future accesses will not trigger `__getattr__`. In contrast, `acct` is stored as `_acct` so that later accesses to `acct` do invoke `__getattr__`.

For comparison purposes, this alternative comes in at 34 lines of code—5 fewer than the property-based version and 11 fewer than the version using descriptors (though replacing `if` with `match` here added two lines, along with extra indentation). Clarity matters more than code size, of course, but extra code can imply extra development and maintenance work. Probably more important here are *roles*: generic tools like `__getattr__` are better suited to generic delegation, while properties and descriptors are designed to manage specific attributes.

Also note again that the code here incurs *extra calls* when setting unmanaged attributes (e.g., `addr`), although no extra calls are incurred for fetching unmanaged attributes since they are defined. Though this will likely result in negligible overhead for most programs, the more narrowly focused properties and descriptors incur an extra call only when managed attributes are accessed, and also appear in `dir` results automatically when needed by generic tools.

Example 38-24. validate_getattr.py

```
class CardHolder:
    acctlen = 8                               # Class data
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                      # Instance data
        self.name = name                     # These trigger __setattr__ too
        self.age = age                       # _acct not mangled: name tested
        self.addr = addr                     # addr is not managed
                                           # remain has no data

    def __getattr__(self, name):
        match name:
            case 'acct':
                return self._acct[:-3] + '***'    # On undefined attr fetches
                                                    # name, age, addr are defined
            case 'remain':
                return self.retireage - self.age    # Doesn't trigger __getattr__
            case _:
```

```

        raise AttributeError(name)

def __setattr__(self, name, value):
    match name:
        case 'name':
            value = value.lower().replace(' ', '_') # On all attr assignments # addr stored directly
        case 'age':
            if value < 0 or value > 150: # acct mangled to _acct
                raise ValueError('invalid age')
        case 'acct':
            name = '_acct'
            value = value.replace('-', '')
            if len(value) != self.acctlen:
                raise TypeError('invalid acct number')
        case 'remain':
            raise TypeError('cannot set remain')
    self.__dict__[name] = value # Avoid looping (or object)

```

When this code is run with either test script, it produces the same output (with a different class name):

```

$ python3 validate_tester.py validate_getattr
...same output as properties, except class name...

$ python3 validate_tester_plus.py validate_getattr
...same output as instance-state descriptors, except class name...

```

Using `__getattribute__` to Validate

Our final variant uses the `__getattribute__` catchall to intercept attribute fetches and manage them as needed. Every attribute fetch is caught here, so we test the attribute names to detect managed attributes and route all others to the superclass for normal fetch processing. This version uses the same `__setattr__` to catch assignments as the prior version (there is no corresponding “`__setattribute__`” in Python—so far?).

Example 38-25 codes this last mod. It works very much like the `__getattr__` version, so we won’t repeat the full description here. Note, though, that because *every* attribute fetch is routed to `__getattribute__`, we don’t need to mangle names to intercept them here (`acct` is stored as `acct`). On the other hand, this code must take care to route nonmanaged attribute fetches to a superclass to avoid looping or extra calls.

Also, notice that this version incurs extra calls for both setting and fetching unmanaged attributes (e.g., `addr`); if speed is paramount, this alternative may be the slowest of the bunch. For comparison purposes, this version amounts to 34 lines of code, just like the prior version (and again including 2 lines added by `match`).

Example 38-25. `validate_getattribute.py`

```

class CardHolder:
    acctlen = 8 # Class data
    retireage = 62.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct # Instance data
        self.name = name # These trigger __setattr__ too
        self.age = age # acct not mangled: name tested
        self.addr = addr # addr is not managed
                        # remain has no data

```

```

def __getattr__(self, name):
    superget = object.__getattr__
    match name:
        case 'acct':
            return superget(self, 'acct')[:-3] + '***'
        case 'remain':
            return superget(self, 'retireage') - superget(self, 'age')
        case _:
            return superget(self, name)

def __setattr__(self, name, value):
    match name:
        case 'name':
            value = value.lower().replace(' ', '_')
        case 'age':
            if value < 0 or value > 150:
                raise ValueError('invalid age')
        case 'acct':
            value = value.replace('-', '')
            if len(value) != self.acctlen:
                raise TypeError('invalid acct number')
        case 'remain':
            raise TypeError('cannot set remain')
    self.__dict__[name] = value

```

Both the `__getattr__` and `__getattribute__` scripts work the same as the property and per-client-instance descriptor versions when run by both tester scripts—*four ways to achieve the same goal in Python*, though they vary in structure and are perhaps less redundant in some other roles:

```

$ python3 validate_tester.py validate_getattribute
...same output as properties, except class name...

$ python3 validate_tester_plus.py validate_getattribute
...same output as instance-state descriptors, except class name...

```

Be sure to study and run this section's code on your own for more pointers on managed-attribute coding techniques.

Chapter Summary

This chapter covered the various techniques for managing access to attributes in Python, including the `__getattr__` and `__getattribute__` operator-overloading methods, and class properties and descriptors. Along the way, it compared and contrasted these tools and presented a handful of use cases to demonstrate their behavior.

Chapter 39 continues our tool-building focus with a survey of *decorators*—code run automatically at function and class creation time rather than on attribute access. Before we continue, though, let's work through a set of questions to review what we've covered here.

Test Your Knowledge: Quiz

1. How do `__getattr__` and `__getattribute__` differ?
2. How do properties and descriptors differ?
3. How are properties and decorators related?

4. What are the main functional differences between `__getattr__` and `__getattribute__` and properties and descriptors?

Test Your Knowledge: Answers

1. The `__getattr__` method is run for explicit fetches of *undefined* attributes only (i.e., those not present on an instance and not inherited from any of its classes). By contrast, the `__getattribute__` method is called for *every* explicit attribute fetch, whether the attribute is defined or not. Because of this, code inside a `__getattr__` can freely fetch other attributes if they are defined, whereas `__getattribute__` must use special code for all such attribute fetches to avoid looping or extra calls (it must route fetches to a superclass to skip itself). Neither method is run for *implicit* fetches of built-in operations (sans the next chapter's heroics).
2. Properties serve a specific role, whereas descriptors are more general. Properties define get, set, and delete functions for a specific attribute; descriptors provide a class with methods for these actions, too, but they provide extra flexibility to support more arbitrary actions. In fact, properties are really a simple way to create a specific kind of descriptor—one that runs functions on attribute accesses. Coding differs too: a property is created with a built-in function, and a descriptor is coded with a class; thus, descriptors can leverage all the usual OOP features of classes, such as inheritance. Moreover, in addition to the instance's state information, descriptors have local state of their own, which can sometimes avoid name collisions in the instance.
3. Properties can be coded with decorator syntax. Because the property built-in accepts a single function argument and returns a function, it can be used directly as a function decorator to define a fetch-access property. Due to the name rebinding behavior of decorators, the name of the decorated function is assigned to a property whose get accessor is set to the original function decorated (`name=property(name)`). Property setter and deleter attributes allow us to further add set and delete accessors with decoration syntax—they set the accessor to the decorated function and return the augmented property. Some may find this a bit clumsy, but this is subjective.
4. The `__getattr__` and `__getattribute__` methods are more generic: they can be used to catch arbitrarily many attributes. In contrast, each property or descriptor provides access interception for only one *specific* attribute—we can't catch every attribute fetch with a single property or descriptor. On the other hand, properties and descriptors handle both attribute fetch and *assignment* by design: `__getattr__` and `__getattribute__` handle fetches only; to intercept assignments as well, `__setattr__` must also be coded. The implementation is also different: `__getattr__` and `__getattribute__` are operator-overloading methods, whereas properties and descriptors are objects manually assigned to class attributes. Unlike the others, properties and descriptors can also sometimes avoid extra calls on assignment to unmanaged names and show up in `dir` results automatically, but are also narrower in scope—they can't address generic delegation goals. In Python evolution, new features tend to offer alternatives but often do not fully subsume what came before.

Decorators

In [Chapter 32](#)'s survey of class odds and ends, we met properties and static and class methods, took a quick look at the `@` decorator syntax Python offers for declaring them, and previewed decorator coding techniques. We also met function decorators briefly while exploring the property built-in in [Chapter 38](#), in the context of abstract superclasses in [Chapter 29](#), and in capsule form in [Chapter 19](#).

This chapter picks up where all this previous decorator coverage left off. Here, we'll dig deeper into the mechanics of decorators and study more ways to code new decorators ourselves with tools like arguments and nesting. As we'll find, other concepts we studied earlier—especially state retention—show up regularly in decorators.

This is a somewhat advanced topic, and decorator construction tends to be of more interest to tool builders than to application programmers. Still, given that decorators are becoming increasingly common in popular Python frameworks, a basic understanding can help demystify their role, even if you're just a decorator user.

Besides covering decorator construction details, this chapter serves as a more realistic *case study* of Python in action. Because its examples grow larger than many of the others we've seen in this book, they better illustrate how code comes together into more complete systems and tools. As an extra perk, some of the code we'll write here may be used as general-purpose tools in your day-to-day programs.

What's a Decorator?

Simply put, *decoration* is a way to specify management or augmentation code for functions and classes. Decorators themselves take the form of callable objects (e.g., functions) that process other callable objects. As suggested earlier in this book, Python decorators come in two related flavors:

- *Function decorators*, added first, do name rebinding at function definition time, providing a layer of logic that can manage functions and methods or later calls to them.
- *Class decorators*, added later, do name rebinding at class definition time, providing a layer of logic that can manage classes or the instances created by later calls to them.

In short, decorators provide a way to insert *automatically run code* at the close of function and class definition statements—at the end of a `def` for function decorators and at the end of a `class` for class decorators. Such code can play a variety of roles, as described in the following sections.

Managing Calls and Instances

In typical use, this automatically run code may be used to augment calls to functions and classes. It arranges this by installing *wrapper* (a.k.a. *proxy*) objects to be invoked later:

Call proxies

Function decorators install wrapper objects to intercept later *function calls* and process them as needed, usually passing the call on to the original function to run the managed action.

Interface proxies

Class decorators install wrapper objects to intercept later *instance-creation calls* and process them as required, usually passing the call on to the original class to create a managed instance.

Decorators achieve these effects by automatically rebinding function and class names to other callables at the end of `def` and `class` statements. When later invoked, these callables can perform tasks such as tracing and timing function calls, managing access to class instance attributes, and so on.

Managing Functions and Classes

Although most examples in this chapter deal with using wrappers to intercept later calls to functions and classes, this is not the only way decorators can be used:

Function managers

Function decorators can also be used to manage *function objects* instead of or in addition to later calls to them—to register a function to an API, for instance. Our primary focus here, though, will be on their more commonly used call-wrapper application.

Class managers

Class decorators can also be used to manage *class objects* directly, instead of or in addition to instance-creation calls—to augment a class with new methods or data, for example. Because this role intersects strongly with that of *metaclasses*, we'll explore additional decorator use cases in the next chapter. As detailed there, both tools run at the end of the class creation process, but class decorators often offer a lighter-weight solution.

In other words, function decorators can be used to manage both function calls and function objects, and class decorators can be used to manage both class instances and classes themselves. By returning the decorated object itself instead of a wrapper, decorators become a simple post-creation step for functions and classes.

Regardless of the role they play, decorators provide a convenient and explicit way to code tools useful both during program development and in live production systems.

Using and Defining Decorators

Depending on your job description, you might encounter decorators as a user or a provider. As we've seen, Python itself comes with built-in decorators that have specialized roles—static and class method declaration, property creation, and more. In addition, many popular Python toolkits include decorators to perform tasks such as managing database or user-interface logic. In such cases, we can get by without knowing how the decorators are coded.

For more general tasks, programmers can code arbitrary decorators of their own. For example, function decorators may be used to augment functions with code that adds call tracing or logging, caches call results, performs argument validity testing during debugging, times calls made to functions for optimization, and so on. Any behavior you can imagine adding to—really, wrapping around—a function call is a candidate for custom function decorators.

On the other hand, function decorators are designed to augment only a specific function or method call, not an entire *object interface*. Class decorators fill the latter role better—because they can intercept instance-creation calls, they can be used to implement arbitrary object interface augmentation or management tasks. For example, custom class decorators can trace, validate, or otherwise augment every attribute reference made for an object. They can also be used to implement proxy objects, singleton classes, and other common coding patterns. In fact, you’ll find that many class decorators are a prime application of the *delegation* coding pattern we met in [Chapter 31](#).

Why Decorators?

Like many advanced Python tools, decorators are never required from a purely technical perspective: we can often implement their functionality instead using simple helper function calls or other techniques. And at a base level, we can always manually code the name rebinding that decorators perform automatically.

That said, decorators provide an explicit syntax for such tasks, which makes intent clearer, can minimize augmentation code redundancy, and may help ensure correct API usage:

- Decorators have a very *explicit* syntax, which makes them easier to spot than helper function calls that may be arbitrarily far removed from the subject functions or classes.
- Decorators are applied *once* when the subject function or class is defined; it’s not necessary to add extra code at every call to the class or function, which may have to be changed in the future.
- Because of both of the prior points, decorators make it less likely that a user of an API will *forget* to augment a function or class according to API requirements.

In other words, beyond their technical model, decorators offer some advantages in terms of both code maintenance and consistency. Moreover, as structuring tools, decorators naturally foster *encapsulation* of code, which reduces redundancy and makes future changes easier.

Like most tools, decorators have some potential *drawbacks*, too—when they insert wrapper logic, they can alter the types of the decorated objects, and they may incur extra calls when used as call or interface proxies. On the other hand, the same considerations apply to any technique that adds wrapping logic to objects.

We’ll explore these trade-offs in the context of real code later in this chapter. Although the choice to use decorators is ultimately subjective, their advantages are compelling enough to have escalated them to common practice in the Python world. To help you decide for yourself, let’s turn to the details.

Decorators Versus Macros

Python’s decorators bear similarities to what some call *aspect-oriented programming* in other languages—code inserted to run automatically before or after a function call runs. Their syntax also very closely resembles (and is likely borrowed from) Java’s *annotations*, though Python’s model may be considered more flexible and general.

Some liken decorators to *macros* too, but this isn’t entirely apt and can be misleading. Macros, like C’s `#define` preprocessor directive, are associated with textual replacement and expansion and designed for generating code. By contrast, Python’s decorators are a *runtime* operation based upon name rebinding, callable objects, and often, proxies. While the two may have use cases that sometimes overlap, decorators and macros are fundamentally different in scope, implementation, and coding patterns. Comparing the two seems akin to comparing Python’s `import` with a C `#include`, which similarly confuses a runtime object-based operation with text insertion.

Of course, the term *macro* has also been diluted over time—to some, it now can also refer to any canned series of steps or procedure—and users of other languages might find the analogy to decorators useful anyhow. But they should also keep in mind that decorators are about callable *objects* managing callable *objects*, not text expansion. Python tends to be best understood and used in terms of Python idioms.

The Basics

Let’s get started with a first-pass look at decoration behavior from an abstract perspective. We’ll write real and more substantial code soon, but since most of the magic of decorators boils down to an automatic rebinding operation, it’s important to understand this mapping first—for both functions and classes.

Function Decorator Basics

As previewed earlier in this book, function decorators are largely just syntactic “sugar” that runs one function through another at the end of a `def` statement and rebinds the original function name to the result.

Usage

A function decorator is a sort of *runtime declaration* about the function whose definition follows. The decorator is coded on a line just before the `def` statement that defines a function or method, and it consists of the `@` symbol followed by a reference to a *metafunction*—a function (or other callable object) that manages another function. As of Python 3.9, the code after the `@` can be any *expression* returning a metafunction, but it’s usually a simple name.

In terms of code, function decorators automatically map the following syntax:

```
@decorator          # Decorate function
def F(arg):
    ...
F(99)                # Call function
```

into this equivalent form, where `decorator` is a one-argument callable object that returns a callable object with the same number of arguments as `F`, if not `F` itself:

```
def F(arg):
    ...
F = decorator(F)          # Rebind function name to decorator result

F(99)                     # Essentially calls decorator(F)(99)
```

This automatic name rebinding works on any `def` statement, whether it's for a simple *function* or a *method* within a class. When the function `F` is later called, it's actually calling the object *returned* by the decorator, which may be either another object that implements required wrapping logic or the original function itself.

In other words, decoration essentially maps the first of the following into the second—though the decorator is really run only once, at decoration time:

```
func(6, 7)
decorator(func)(6, 7)
```

This automatic name rebinding accounts for the static-method and property decoration syntax we met earlier in the book:

```
class C:
    @staticmethod
    def meth(...): ...          # meth = staticmethod(meth)

class C:
    @property
    def name(self): ...         # name = property(name)
```

In both cases, the method name is rebound to the result of a built-in function decorator at the end of the `def` statement. Calling the original name later invokes whatever object the decorator returns. In these specific cases, the original names are rebound to a static-method router and property descriptor, but the process is much more general than this—as the next section explains.

Implementation

A decorator itself is a *callable that returns a callable*. That is, it returns the object to be called later when the decorated function is invoked through its original name—either a wrapper object to intercept later calls or the original function augmented in some way. In fact, decorators can *be* any type of callable and *return* any type of callable: any combination of functions and classes may be used, though some are better suited to certain contexts.

For example, to tap into the decoration protocol in order to manage a function just after it is created, we might code a decorator of this form:

```
def decorator(F):
    # Process function F here
    return F

@decorator
def func(): ...               # func = decorator(func)
```

Because the original decorated function is assigned back to its name, this simply adds a post-creation step to function definition. Such a structure might be used to register a function to an API, initialize function attributes, and so on.

In more typical use, to insert logic that intercepts later calls to a function, we might code a decorator to return a different object than the original function—a *proxy* for later calls:

```
def decorator(F):
    # Save or use function F
    # Return a different callable: nested def, class instance with __call__, etc.

@decorator
def func(): ...                # func = decorator(func)
```

This decorator is invoked at decoration time, and the callable it returns is invoked when the original function name is later called. The decorator itself receives the decorated function; the callable returned receives whatever arguments are later passed to the decorated function's name. When coded properly, this works the same for class-level *methods*: the implied instance object simply shows up in the first argument of the returned callable.

In skeleton terms, here's one common coding pattern that captures this idea—the decorator returns a wrapper that retains the original function in an enclosing scope:

```
def decorator(F):                # On @ decoration
    def wrapper(*args):          # On wrapped function call
        # Use F and args
        # F(*args) calls original function
    return wrapper

@decorator
def func(x, y):                  # func = decorator(func)
    ...                          # func is passed to decorator's F

func(6, 7)                      # 6, 7 are passed to wrapper's *args
```

When the name `func` is later called, it really invokes the wrapper function returned by `decorator`; the wrapper function can then run the original `func` because it is still available in an *enclosing scope*. When coded this way, each decorated function produces a new scope to retain state.

To do the same with *classes*, we can overload the call operation and use instance attributes instead of enclosing scopes:

```
class decorator:
    def __init__(self, func):    # On @ decoration
        self.func = func
    def __call__(self, *args):    # On wrapped function call
        # Use self.func and args
        # self.func(*args) calls original function

@decorator
def func(x, y):                 # func = decorator(func)
    ...                         # func is passed to __init__

func(6, 7)                     # 6, 7 are passed to __call__'s *args
```

When the name `func` is later called now, it really invokes the `__call__` operator-overloading method of the instance created by `decorator`; the `__call__` method can then run the original `func` because it is still available in an *instance attribute*. When coded this way, each decorated function produces a new instance to retain state.

Supporting method decoration

One subtle point about the prior class-based coding is that while it works to intercept simple *function* calls, it does not quite work when applied to class-level *method* functions:

```
class decorator:
    def __init__(self, func):          # func is method without instance
        self.func = func
    def __call__(self, *args):         # self is decorator instance
        # self.func(*args) fails!    # C instance not in args!

class C:
    @decorator
    def method(self, x, y):            # method = decorator(method)
        ...                           # Rebound to decorator instance
```

When coded this way, the decorated method is rebound to an *instance* of the decorator class instead of a simple function.

The problem with this is that the `self` in the decorator's `__call__` receives the decorator class instance when the method is later run, and the instance of class `C` is never included in `*args`. This makes it impossible to dispatch the call to the original method—the decorator object retains the original method function, but it has no instance to pass to it.

To support *both* functions and methods, the nested function alternative works better:

```
def decorator(F):
    def wrapper(*args):
        # F(*args) runs func or method
    return wrapper

@decorator
def func(x, y):
    ...
func(6, 7)

class C:
    @decorator
    def method(self, x, y):
        ...

X = C()
X.method(6, 7)
```

When coded this way, `wrapper` receives the `C` class instance in its first argument, so it can dispatch to the original method and access state information.

Technically, this nested-function version works because Python creates a *bound method* object and thus passes the subject class instance to the `self` argument only when a method attribute references a simple function; when it references an instance of a callable class instead, the callable class's instance is passed to `self` to give the callable class access to its own state information. You'll see how this subtle difference can matter in more realistic examples later in this chapter.

Also note that nested functions are perhaps the most straightforward way to support decoration of both functions and methods, but not necessarily the only way. The prior chapter's *descriptors*, for example, receive both the descriptor-class and subject-class instance when called. Though more complex, later in this chapter you'll see how this tool can be leveraged in this context as well.

Class Decorator Basics

Function decorators proved so useful that the model was extended to allow class decoration. They were initially resisted because of role overlap with the next chapter's *metaclasses*; in the end, though, they were adopted because they provide a simpler way to achieve many of the same goals.

Class decorators are strongly related to function decorators; in fact, they use the same syntax and very similar coding patterns. Rather than wrapping individual functions or methods, though, class decorators are a way to manage classes or wrap up instance-creation calls with extra logic that manages or augments instances created from a class. In the latter role, they may manage full object *interfaces* instead of a single callable object.

Usage

Syntactically, class decorators appear just before `class` statements, in the same way that function decorators appear just before `def` statements. In symbolic terms, for a decorator that must be a one-argument callable that returns a callable, the class decorator syntax:

```
@decorator                # Decorate class
class C:
    ...

x = C(99)                  # Make an instance
```

is equivalent to the following—the class is automatically passed to the decorator function, and the decorator's result is assigned back to the class name:

```
class C:
    ...
C = decorator(C)           # Rebind class name to decorator result

x = C(99)                  # Essentially calls decorator(C)(99)
```

The net effect is that calling the class name later to create an instance winds up triggering the callable returned by the decorator, which may or may not call the original class itself.

Implementation

New class decorators are coded with many of the same techniques used for function decorators, though some may involve *two levels* of augmentation—to manage both instance-construction calls as well as instance-interface access. Because a class decorator is also a *callable that returns a callable*, most combinations of functions and classes suffice.

However it's coded, the decorator's result is what runs when an instance is later created. For example, to simply manage a class just after it is created, return the original class itself:

```
def decorator(C):
    # Process class C here
    return C

@decorator
class C: ...                # C = decorator(C)
```

To instead insert a wrapper layer that intercepts later instance-creation calls, return a different callable object:


```

def decorator(C):
    # Save or use class C
    # Return a different callable: nested def, class instance with __call__, etc.

    @decorator
    class C: ...

```

The callable returned by such a class decorator typically creates and returns a new instance of the original class, augmented in some way to manage its interface. For example, the following inserts an object that intercepts undefined attributes of a class instance:

```

def decorator(cls):
    class Wrapper:
        def __init__(self, *args):
            self.wrapped = cls(*args)
        def __getattr__(self, name):
            return getattr(self.wrapped, name)
    return Wrapper

@decorator
class C:
    def __init__(self, x, y):
        self.attr = 'hack'

x = C(6, 7)
print(x.attr)

```

In this example, the decorator rebinds the class name to another class, which retains the original class in an enclosing scope and creates and embeds an instance of the original class when it's called. When an attribute is later fetched from the instance, it is intercepted by the wrapper's `__getattr__` and delegated to the embedded instance of the original class. Moreover, each decorated class creates a new scope, which remembers the original class. We'll flesh out this example into some more useful code later in this chapter.

Like function decorators, class decorators are commonly coded as either *closure* (a.k.a. “factory”) functions that create and return callables, classes that use `__init__` or `__call__` methods to intercept call operations, or some combination thereof. Closure functions typically retain state in enclosing-scope references, and classes retain state in attributes.

Supporting multiple instances

As for function decorators, some callable-type combinations work better for class decorators than others. Consider the following *invalid* alternative to the class decorator of the prior example:

```

class Decorator:
    def __init__(self, C):
        self.C = C
    def __call__(self, *args):
        self.wrapped = self.C(*args)
        return self
    def __getattr__(self, attrname):
        return getattr(self.wrapped, attrname)

@Decorator
class C: ...

x = C()
y = C()

```

This code handles multiple decorated classes (each makes a new `Decorator` instance) and will intercept instance-creation calls (each runs `__call__`). Unlike the prior version, however, this version fails to handle *multiple instances* of a given class—each instance-creation call overwrites the prior saved instance. The prior version does support multiple instances because each instance-creation call makes a new independent wrapper object. More generally, either of the following patterns supports multiple wrapped instances:

```
def decorator(C):                                     # On @ decoration
    class Wrapper:
        def __init__(self, *args):                   # On instance creation: new Wrapper
            self.wrapped = C(*args)                  # Embed instance in instance
    return Wrapper

class Wrapper: ...
def decorator(C):                                     # On @ decoration
    def onCall(*args):                               # On instance creation: new Wrapper
        return Wrapper(C(*args))                    # Embed instance in instance
    return onCall
```

We'll study this phenomenon in a more realistic context later in the chapter too; in practice, though, we must be careful to combine callable types properly to support our intent and choose state policies wisely.

Decorator Nesting

Sometimes, one decorator isn't enough. For instance, suppose you've coded *two* function decorators to be used during development—one to test argument types before function calls and another to test return value types after function calls. You can use either independently, but what to do if you want to employ both on a single function? What you really need is a way to *nest* the two, such that the result of one decorator is the function decorated by the other. It's irrelevant which is nested, as long as both steps run on later calls.

To support multiple nested steps of augmentation this way, decorator syntax allows you to add multiple layers of wrapper logic to a decorated function or method. When this feature is used, each decorator must appear on a line of its own. Decorator syntax of this form:

```
@A
@B
@C
def f(...):
    ...
```

runs the same as the following:

```
def f(...):
    ...
    f = A(B(C(f)))
```

Here, the original function is passed through three different decorators, and the resulting callable object is assigned back to the original name. Each decorator processes the result of the prior, which may be the original function or an inserted wrapper.

If all the decorators insert wrappers, the net effect *stacks* them: when the original function name is called, three different layers of wrapping object logic will be invoked to augment the original function in three different ways. The last decorator listed is the first applied and, thus, the most deeply nested when the original function name is later called.

Just as for functions, multiple class decorators result in multiple nested function calls and possibly multiple levels and steps of wrapper logic around instance-creation calls. For example, the following code:

```
@hack
@code
class C:
    ...

X = C()
```

is equivalent to the following:

```
class C:
    ...
    C = hack(code(C))

X = C()
```

Again, each decorator is free to return either the original class or an inserted wrapper object. With wrappers, when an instance of the original C class is finally requested, the call is redirected to the wrapping layer objects provided by both the hack and code decorators, which may have arbitrarily different roles—they might trace and validate attribute access for example, and both steps would be run in turn on later requests.

For instance, the following do-nothing decorators simply return the decorated function:

```
def d1(F): return F
def d2(F): return F
def d3(F): return F

@d1
@d2
@d3
def func():                # func = d1(d2(d3(func)))
    print('hack')

func()                     # Prints "hack"
```

The same syntax works on classes, as do these same do-nothing decorators.

When decorators insert wrapper function objects, though, they may augment the original function when called—the following concatenates to its result in the decorator layers, as it runs the layers from inner to outer:

```
def d1(F): return lambda: 'X' + F()
def d2(F): return lambda: 'Y' + F()
def d3(F): return lambda: 'Z' + F()

@d1
@d2
@d3
def func():                # func = d1(d2(d3(func)))
    return 'hack'

print(func())              # Prints "XYZhack"
```

We use lambda functions to implement wrapper layers here (each retains the wrapped function F in an enclosing scope); in practice, wrappers can take the form of functions, callable classes, and more. When designed well, decorator nesting allows us to combine augmentation steps in a wide variety of ways.

Decorator Arguments

Both function and class decorators can also seem to take *arguments*. Really, though, the role of these arguments is simpler than it may seem: decorator arguments are passed to a callable that *returns* the decorator—which in turn returns a callable. By nature, this usually sets up multiple levels of state retention. The following, for instance:

```
@decorator(A, B)
def F(arg):
    ...

F(99)
```

is automatically mapped into this equivalent form, where `decorator` is a callable that *returns* the actual decorator. The returned decorator in turn returns the callable run later for calls to the original function name:

```
def F(arg):
    ...
F = decorator(A, B)(F)    # Rebind F to result of decorator's return value

F(99)                    # Essentially calls decorator(A, B)(F)(99)
```

Decorator arguments are resolved *before* decoration ever occurs, and they are usually used to retain state information for use in later calls. The decorator function in this example, for instance, might take a form like the following:

```
def decorator(A, B):
    # Save or use A, B
    def actualDecorator(F):
        # Save or use function F
        # Return a callable: nested def, class instance with __call__, etc.
        return callable
    return actualDecorator
```

The outer function in this structure generally saves the decorator arguments away as state information for use in the actual decorator, the callable it returns, or both. This code snippet retains the state information argument in enclosing function scope references, but class attributes would work as well.

In other words, decorator arguments often imply *three levels of callables*: a callable to accept decorator arguments, which returns a callable to serve as decorator, which returns a callable to handle calls to the original function or class. Each of the three levels may be a function or class and may retain state in the form of scopes or class attributes.

Decorator arguments can be used to provide attribute initialization values, call-trace message labels, attribute names to be validated, and much more—any sort of configuration *parameter* for objects or their proxies is a candidate. We'll code concrete examples of decorator arguments later in this chapter.

Decorators Manage Functions and Classes, Too

To wrap up, although much of the rest of this chapter focuses on wrapping later calls to functions and classes, it's important to remember that the decorator mechanism is more general than this—it is simply a protocol for passing functions and classes through any callable immediately after they are created. As such, it can also be used to invoke arbitrary post-creation processing:

```

def decorator(0):
    # Augment function or class 0
    return 0

@decorator
def F(): ...           # F = decorator(F)

@decorator
class C: ...           # C = decorator(C)

```

If we return the original decorated object this way instead of a proxy, we can manage functions and classes *themselves* rather than later calls to them. Such decorators might be used to register callable objects to an API, initialize attributes in functions or classes when they are created, and so on. Decorator roles are limited only by your imagination.

Coding Function Decorators

On to the code. In the rest of this chapter, we are going to study working examples that demonstrate the decorator concepts we just surveyed. This section presents a handful of function decorators in complete form, and the next shows tangible class decorators in action. Following that, we'll close out with two larger case studies that showcase typical decorator roles and code full-scale implementations of class privacy and argument range tests.

Tracing Function Calls

To get started, let's revive the call tracer example we met in [Chapter 32](#). [Example 39-1](#) defines and applies a function decorator that counts the number of calls made to the decorated function and prints a trace message for each call.

Example 39-1. decorator1.py

```

class tracer:
    def __init__(self, func):          # On @ decoration: save original func
        self.calls = 0
        self.func = func
    def __call__(self, *args):         # On later calls: run original func
        self.calls += 1
        print(f'call {self.calls} to {self.func.__name__}')
        self.func(*args)

@tracer
def hack(a, b, c):                   # hack = tracer(hack)
    print(a + b + c)                 # Wraps hack in a decorator object

```

Notice how each function decorated with this class will create a new instance with its own saved function object and calls counter. Also, observe how the `*args` argument syntax is used to pack and unpack arbitrarily many passed-in arguments. This generality enables this decorator to be used to wrap any function with any number of positional arguments; this version doesn't yet work on keyword arguments or class-level methods and doesn't return results, but we'll fix these shortcomings later in this section.

Now, if we import this module's function and test it interactively in a REPL, we get the following sort of behavior—each call generates a trace message initially because the decorator class intercepts it:

```
$ python3
>>> from decorator1 import hack

>>> hack(1, 2, 3)           # Really calls the tracer wrapper object
call 1 to hack
6

>>> hack('a', 'b', 'c')    # Invokes __call__ in class
call 2 to hack
abc

>>> hack.calls              # Number calls in wrapper state information
2
>>> hack
<decorator1.tracer object at 0x10cafc680>
```

When run, the tracer class saves away the decorated function and intercepts later calls to it in order to add a layer of logic that counts and prints each call. Notice how the total number of calls shows up as an attribute of the decorated function—`hack` is really an instance of the tracer class when decorated, a finding that may have ramifications for programs that do type checking, but is generally benign.

For function calls, the `@` decoration syntax can be more convenient than modifying each call to account for the extra logic level, and it avoids accidentally calling the original function directly. Consider a *non-decorator* equivalent such as the following:

```
>>> calls = 0
>>> def tracer(func, *args):
    global calls
    calls += 1
    print(f'call {calls} to {func.__name__}')
    func(*args)

>>> def hack(a, b, c):      # Nondecorated function
    print(a, b, c)

>>> hack(1, 2, 3)          # Normal nontraced call: accidental?
1 2 3
>>>
>>> tracer(hack, 1, 2, 3)   # Special traced call without decorators
call 1 to hack
1 2 3
```

This alternative can be used on any function without the special `@` syntax, but unlike the decorator version, it requires extra syntax at every place where the function is *called* in your code. Furthermore, its intent may not be as obvious, and it does not ensure that the extra layer will be invoked for normal calls. Although decorators are never *required* (we can always rebind names manually), they are often the most convenient and uniform augmentation option.

Decorator State Retention Options

The preceding example raises an important point. Decorators have a variety of options for retaining state information provided at decoration time to be used during later calls to decorated objects. They generally need to support multiple decorated objects and multiple later calls, but there are several ways

to implement these goals: instance attributes, global variables, nonlocal closure variables, and function attributes can all be used for retaining state.

This topic parallels the initial state coverage in [Chapter 17](#) but can be fleshed out here with class details, and it is so endemic to decorators that it qualifies as a prerequisite. This topic also applies to both function and class decorators, but let's explore it in the narrower function-decorator realm.

State with class-instance attributes

As an opening act in the state-retention show, [Example 39-2](#) codes an augmented version of the prior example, which adds support for *keyword* arguments with ****** syntax and *returns* the wrapped function's result to support more use cases (for nonlinear readers, we first studied keyword arguments in [Chapter 18](#)).

Example 39-2. decorator_state_classes.py

```
class tracer:
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.calls += 1
        print(f'call {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)

@tracer
def hack(a, b, c):
    print(a + b + c)

@tracer
def code(x, y):
    print(x ** y)

if __name__ == '__main__':
    hack(1, 2, 3)
    hack(a=4, b=5, c=6)

    code(4, 2)
    code(2, y=16)
```

Like the original, this uses *class instance attributes* to save state explicitly. Both the wrapped function and the calls counter are *per-instance* information—each decoration gets its own copy. When run as a script, the output of this version is as follows; notice how the *hack* and *code* functions each have their own calls counter because each decoration creates a new class instance:

```
$ python3 decorator_state_classes.py
call 1 to hack
6
call 2 to hack
15
call 1 to code
16
call 2 to code
65536
```

While useful for decorating functions, this coding scheme still has issues when applied to *methods*—a shortcoming we'll address in a later revision.

State with global variables

For simpler tasks that don't require per-function data, moving state variables out to the *global scope*, as illustrated by [Example 39-3](#), might suffice. This code still uses an enclosing-scope reference for the original decorated function but pushes the call counter out to the enclosing module.

Example 39-3. *decorator_state_globals.py*

```
calls = 0
def tracer(func):
    def wrapper(*args, **kwargs):
        global calls
        calls += 1
        print(f'call {calls} to {func.__name__}')
        return func(*args, **kwargs)
    return wrapper

@tracer
def hack(a, b, c):
    print(a + b + c)

@tracer
def code(x, y):
    print(x ** y)

if __name__ == '__main__':
    hack(1, 2, 3)
    hack(a=4, b=5, c=6)

    code(4, 2)
    code(2, y=16)
```

Unfortunately, moving the counter out to the common global scope to allow it to be changed like this also means that it will be *shared* by every wrapped function. Unlike class instance attributes, global counters are cross-program, not per-function—the counter is incremented for *any* traced function call. You can tell the difference if you compare this version's output with the prior version's—the single, shared global call counter is incorrectly updated by calls to every decorated function:

```
$ python3 decorator_state_globals.py
call 1 to hack
6
call 2 to hack
15
call 3 to code
16
call 4 to code
65536
```

State with enclosing-scope nonlocals

Shared global state may be what we want in some cases. If we really want a *per-function* counter, though, we can either use classes as before or make use of *closure* functions and the `nonlocal` statement described in [Chapter 17](#). Because this statement allows enclosing function scope variables to be *changed*, they can serve as per-decoration, changeable data. [Example 39-4](#) demos the basics of this scheme.

Example 39-4. *decorator_state_nonlocals.py*

```
def tracer(func):                                # State via enclosing scope and nonlocal
    calls = 0                                    # Instead of class attrs or global
    def wrapper(*args, **kwargs):                # calls is per-function, not global
        nonlocal calls
        calls += 1
        print(f'call {calls} to {func.__name__}')
        return func(*args, **kwargs)
    return wrapper

@tracer
def hack(a, b, c):                               # Same as: hack = tracer(hack)
    print(a + b + c)

@tracer
def code(x, y):                                  # Same as: code = tracer(code)
    print(x ** y)

if __name__ == '__main__':
    hack(1, 2, 3)                                # Really calls wrapper, bound to hack
    hack(a=4, b=5, c=6)                          # wrapper calls hack

    code(4, 2)                                    # Really calls wrapper, bound to code
    code(2, y=16)                                # Nonlocal calls _is_ per-decoration here
```

Now, because enclosing-scope variables are not cross-program globals, each wrapped function gets its own counter again, just as for classes and attributes. Here's the new output:

```
$ python3 decorator_state_nonlocals.py
call 1 to hack
6
call 2 to hack
15
call 1 to code
16
call 2 to code
65536
```

State with function attributes

Finally, you can also avoid globals and classes by making use of *function attributes* for some changeable state instead of `nonlocal`. As we saw in Chapters 17 and 19, we can attach arbitrary attributes to functions by assignment, with *func.attr=value*. Because a factory function makes a new function on each call, its attributes become per-call state. Moreover, you need to use this technique only for state variables that must *change*; enclosing-scope references are still retained and work normally.

To demo, [Example 39-5](#) simply uses `wrapper.calls` for state. It works the same as the preceding `nonlocal` version because the counter is again per-decorated-function.

Example 39-5. *decorator_state_attributes.py*

```
def tracer(func):                                # State via enclosing scope and func attr
    def wrapper(*args, **kwargs):                # calls is per-function, not global
        wrapper.calls += 1
        print(f'call {wrapper.calls} to {func.__name__}')
        return func(*args, **kwargs)
    wrapper.calls = 0
```

```

    return wrapper

@tracer
def hack(a, b, c):           # Same as: hack = tracer(hack)
    print(a + b + c)

@tracer
def code(x, y):              # Same as: code = tracer(code)
    print(x ** y)

if __name__ == '__main__':
    hack(1, 2, 3)            # Really calls wrapper, assigned to hack
    hack(a=4, b=5, c=6)     # wrapper calls hack

    code(4, 2)               # Really calls wrapper, assigned to code
    code(2, y=16)            # wrapper.calls _is_ per-decoration here

```

As we learned in [Chapter 17](#), this works only because the name `wrapper` is retained in the enclosing `tracer` function's scope. When we later increment `wrapper.calls`, we are not changing the name `wrapper` itself, so no `nonlocal` declaration is required:

```

$ python3 decorator_state_attributes.py
...same output as prior version...

```

This scheme was almost relegated to a footnote because it may be more obscure than `nonlocal` and might be better saved for cases where other schemes don't help. However, function attributes also have a substantial advantage: like class instances, they allow access to the saved state from *outside* the decorator's code; `nonlocals` can only be seen inside the nested function itself, but function attributes have wider visibility.

We will employ function attributes again in an answer to one of the end-of-chapter questions, where their visibility outside callables becomes an asset. As changeable state associated with a context of use, though, they are equivalent to enclosing-scope `nonlocals`. As usual, choosing from multiple tools is an inherent part of the programming task.

Because decorators often imply multiple levels of callables, you can combine functions with enclosing scopes, classes with attributes, and function attributes to achieve a variety of coding structures. As you'll see later, though, this sometimes may be subtler than you expect—each decorated function should have its own state, and each decorated class may require state both for itself and for each generated instance.

In fact, as the next section will explain in more detail, if we want to apply function decorators to class-level methods, too, we also have to be careful about the distinction Python makes between decorators based on callable class instance objects and decorators based on nested functions.

Class Pitfall: Decorating Methods

When the preceding section's class-based `tracer` function decorator, [Example 39-2](#), was initially coded, it was assumed that it could also be applied to any *method*—decorated methods should work the same, but the automatic `self` instance argument would simply be included at the front of `*args`. The only real downside to this assumption is that it is *completely wrong*, though the reasons for the failure are far from obvious.

In short, when applied to a class's method, this version of the `tracer` fails because `self` is the instance of the decorator class and the instance of the decorated subject class is not included in `*args` at all. Here's a relisting of the class in question to avoid page flipping:

```

class tracer:
    def __init__(self, func):          # State via instance attributes
        self.calls = 0                # On @ decorator
        self.func = func              # Save func for later call
    def __call__(self, *args, **kwargs): # On call to original function
        self.calls += 1
        print(f'call {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)

```

This phenomenon was introduced abstractly earlier in this chapter, but now we can see it in the context of working code. [Example 39-2](#)’s class-based decorator works as advertised earlier for plain functions (copy/pasters: don’t copy the initial “...” REPL prompts included in this chapter to preserve indentation after decorator lines):

```

>>> from decorator_state_classes import tracer
>>> @tracer
... def hack(a, b, c):                # hack = tracer(hack)
    print(a + b + c)                 # Triggers tracer.__init__

>>> hack(1, 2, 3)                     # Runs tracer.__call__
call 1 to hack
6
>>> hack(a=4, b=5, c=6)               # hack saved in an instance attribute
call 2 to hack
15

```

However, decoration of class-level methods fails (more lucid sequential readers might recognize this as an adaptation of our `Person` class resurrected from the object-oriented tutorial in [Chapter 28](#)):

```

>>> class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    @tracer
    def giveRaise(self, percent):      # giveRaise = tracer(giveRaise)
        self.pay *= (1.0 + percent)

>>> pat = Person('Pat Jones', 50_000) # tracer remembers method funcs
>>> pat.giveRaise(.10)                 # Runs tracer.__call__(???, .10)
call 1 to giveRaise
TypeError: Person.giveRaise() missing 1 required positional argument: 'percent'

```

The root of the problem here is in the `self` argument of the tracer class’s `__call__` method—is it a tracer instance or a `Person` instance? We ultimately need *both* as it’s coded: the tracer for decorator state, and the `Person` for routing on to the original method. Really, `self` *must* be the tracer object to provide access to tracer’s state information (its `calls` and `func`); this is true whether decorating a simple function or a method.

Unfortunately, when our decorated method name is rebound to a class instance object with a `__call__`, Python passes only the tracer instance to `self`; it doesn’t pass along the `Person` subject in the arguments list at all. Moreover, because the tracer knows nothing about the `Person` instance we are trying to process with method calls, there’s no way to create a bound method with an instance, and thus, no way to correctly dispatch the call. This isn’t a bug, but it’s wildly subtle.

In the end, the prior listing winds up passing too few arguments to the decorated method, and results in an error. Add a line to the decorator’s `__call__` to print all its arguments to verify this—as you can see, `self` is the tracer instance, and the `Person` instance is entirely absent:

```
>>> pat.giveRaise(.10)
<__main__.tracer object at 0x108a02c00> (0.10,) {}
```

As mentioned earlier, this happens because Python passes the implied subject instance to `self` when a method name is bound to a simple function only; when it is an instance of a callable class, that class's instance is passed instead. That is, Python makes a *bound method* object containing the subject instance *only* when the method is a simple function, not when it is a callable instance of another class.

Using nested functions to decorate methods

If you want your function decorators to work on *both* simple functions and class-level methods, the most straightforward solution lies in using one of the other state retention solutions described earlier—code your function decorator as *nested* `def` statements so that you don't depend on a single `self` instance argument to be both the wrapper class instance and the subject class instance.

In fact, we already *have*—both Examples 39-4 and 39-5 work for both functions and class methods by using nested functions along with nonlocal variables or function attributes:

```
>>> from decorator_state_nonlocals import tracer          # See Example 39-4

>>> @tracer
... def hack(a, b, c):                                   # Works for functions
...     print(a + b + c)

>>> hack(1, 2, 3)
call 1 to hack
6

>>> class Person:                                       # AND works for methods
...     def __init__(self, name, pay):
...         self.name = name
...         self.pay = pay
...     @tracer
...     def giveRaise(self, percent):                   # self included in args
...         self.pay *= (1.0 + percent)                 # Counter in nonlocals

>>> pat = Person('Pat Jones', 50_000)
>>> pat.giveRaise(.10)
call 1 to giveRaise
>>> pat.giveRaise(.10)
call 2 to giveRaise
>>> f'{pat.pay:,.2f}'
'60,500.00'

>>> from decorator_state_attributes import tracer        # See Example 39-5
...same correct results...                               # Counter in attributes
```

Because decorated methods here are rebound to simple functions instead of instance objects, Python correctly passes the `Person` object as the first argument, and the decorator propagates it on in the first item of `*args` to the `self` argument of the real, decorated methods. Trace through these results and decorators to make sure you have a handle on this model; the next section provides an alternative to it that supports classes but is also substantially more complex.

Using descriptors to decorate methods

Although the nested function solution illustrated in the prior section is the most straightforward way to support decorators that apply to both functions and class-level methods, other schemes are possible. The *descriptor* feature we explored in the prior chapter, for example, can help here as well.

Recall from our discussion in the prior chapter that a descriptor is normally a class attribute assigned to an object with a `__get__` method run automatically whenever that attribute is referenced and fetched:

```
class Descriptor:
    def __get__(self, instance, owner): ...

class Subject:
    attr = Descriptor()

X = Subject()
X.attr          # Roughly runs Descriptor.__get__(Subject.attr, X, Subject)
```

Descriptors may also have `__set__` and `__del__` access methods, but we don't need them here. More relevant to this chapter's topic: because the descriptor's `__get__` method receives *both* the descriptor class instance and subject class instance when invoked, it's well suited to decorating methods when we need both the decorator's state and the original class instance for dispatching calls. Consider the alternative tracing decorator in [Example 39-6](#), which *also* happens to be a descriptor when used for a class-level method (its `...` are the same as in the prior REPL session).

Example 39-6. *calltracer_desc_class.py*

```
class tracer(object):
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.calls += 1
        print(f'call {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):
        return wrapper(self, instance)

class wrapper:
    def __init__(self, desc, subj):
        self.desc = desc
        self.subj = subj
    def __call__(self, *args, **kwargs):
        return self.desc(self.subj, *args, **kwargs)

@tracer
def hack(a, b, c):
    ...

class Person:
    ...
    @tracer
    def giveRaise(self, percent):
        ...
```

This works the same as the preceding nested function coding. Its operation varies by usage context:

- Decorated *functions* invoke only its `__call__`, and never invoke its `__get__`.
- Decorated *methods* invoke its `__get__` first to resolve the method name fetch (on *I.method*); the object returned by `__get__` retains the subject class instance and is then invoked to complete the call expression, thereby triggering the decorator's `__call__` (on `()`).

For example, given the same testing code, the call to:

```
pat.giveRaise(.10)                                # Runs __get__ then __call__
```

runs `tracer.__get__` first because the `giveRaise` attribute in the `Person` class has been rebound to a descriptor by the method function decorator. The call expression then triggers the `__call__` method of the returned wrapper object, which in turn invokes `tracer.__call__`. In other words, decorated method calls trigger a *five-step* process: `tracer.__get__`, which invokes `wrapper.__init__`, followed by three call operations—`wrapper.__call__`, `tracer.__call__`, and finally the original wrapped method.

The wrapper object retains both descriptor and subject instances, so it can route control back to the original decorator/descriptor class instance. In effect, the wrapper object saves the subject class instance available during method attribute fetch and adds it to the later call's arguments list, which is passed to the decorator `__call__`. Routing the call back to the descriptor class instance this way is required in this application so that all calls to a wrapped method use the same calls counter state information in the descriptor instance object.

Alternatively, we could use a nested function and enclosing-scope references to achieve the same effect—[Example 39-7](#) works the same as the preceding one by swapping a wrapper class and attributes for a nested function and scope references. It requires noticeably less code but follows a similar multistep process on each decorated method call.

Example 39-7. calltracer_desc_func.py

```
class tracer(object):
    def __init__(self, func):          # On @ decorator
        self.calls = 0                # Save func for later call
        self.func = func
    def __call__(self, *args, **kwargs): # On call to original func
        self.calls += 1
        print(f'call {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner): # On method fetch
        def wrapper(*args, **kwargs): # Retain both inst
            return self(instance, *args, **kwargs) # Runs __call__
        return wrapper
```

...rest same as Example 39-6...

These two descriptor-based tracers work the same as the nested-functions version, so we'll skip their output here. Add `print` statements to their methods to trace their multistep get/call processes if it helps. In either coding, this descriptor-based scheme is also substantially subtler than the nested-function option, and so is probably a second choice here. To be more blunt, if its complexity doesn't send you screaming into the night, its performance costs probably should! Still, this may be a useful coding pattern in other contexts.

Before moving on, it's also worth briefly noting that we might code this descriptor-based decorator more simply as in [Example 39-8](#), but it would then *apply only to methods*, not to simple functions—an intrinsic limitation of attribute descriptors, and just the inverse of the problem we're trying to solve (application to both functions and methods).

Example 39-8. *calltracer_desc_fail.py*

```
class tracer(object):
    def __init__(self, meth):          # For methods, but not functions!
        self.calls = 0                # On @ decorator
        self.meth = meth
    def __get__(self, instance, owner): # On method fetch
        def wrapper(*args, **kwargs): # On method call: proxy with self+inst
            self.calls += 1
            print(f'call {self.calls} to {self.meth.__name__}')
            return self.meth(instance, *args, **kwargs)
        return wrapper

@tracer
def hack(a, b, c):                    # OK for methods but FAILS for functions
    ...                               # hack = tracer(hack)
    ...                               # No attribute fetch occurs on calls!

...rest same as Example 39-6...
```

In the rest of this chapter we're going to be casual about using classes or functions to code our function decorators, as long as they are applied only to functions. Some decorators may not require the instance of the original class, and will still work on both functions and methods if coded as a class—something like Python's own `staticmethod` decorator, for example, wouldn't require an instance of the subject class (indeed, its whole point is to remove the instance from the call).

The simpler moral of this story, though, is that if you want your decorators to work on both simple functions and methods, you're probably better off using the *nested-function* coding pattern instead of a class with call interception.

Timing Function Calls

To better sample what function decorators are capable of, let's turn to a different use case. Our next decorator times *calls* made to a decorated function—both the time for one call and the total time among all calls. As coded in [Example 39-9](#), the decorator is applied to two functions to compare the speeds of list comprehensions and the `map` built-in.

Example 39-9. *timerdeco1.py*

```
"Caveat: timer won't work on methods as coded (see quiz solution)"
import time, sys

class timer:
    def __init__(self, func):
        self.func = func
        self.alltime = 0
    def __call__(self, *args, **kwargs):
        start = time.perf_counter()
        result = self.func(*args, **kwargs)
        elapsed = time.perf_counter() - start
        self.alltime += elapsed
```

```

        print(f'{self.func.__name__}: {elapsed:.5f}, {self.alltime:.5f}')
        return result

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

@timer
def mapcall(N):
    return list(map((lambda x: x * 2), range(N)))

if __name__ == '__main__':
    for func in (listcomp, mapcall):
        result = func(5)                                # Time for this call, result
        func(50_000)
        func(500_000)
        func(1_000_000)
        print(result)
        print(f'allTime = {func.alltime}\n')            # Total time for all func calls

    print('**map/comp =', round(mapcall.alltime / listcomp.alltime, 3))

```

When run on a macOS host by CPython 3.12, the output of this file's self-test code is as follows—giving for each function call the function name, time for this call, and time for all calls so far, along with the first call's return value, cumulative time for each function, and the map-to-comprehension time ratio at the end:

```

$ python3 timerdeco1.py
listcomp: 0.00000, 0.00000
listcomp: 0.00366, 0.00366
listcomp: 0.03134, 0.03500
listcomp: 0.05213, 0.08713
[0, 2, 4, 6, 8]
allTime = 0.08712841104716063

mapcall: 0.00001, 0.00001
mapcall: 0.00396, 0.00397
mapcall: 0.04082, 0.04479
mapcall: 0.07789, 0.12268
[0, 2, 4, 6, 8]
allTime = 0.12268476499593817

**map/comp = 1.408

```

Times vary per Python version, test machine, and other variables, of course, and cumulative time is available as a class instance attribute here. As usual, `map` calls are slower than list comprehensions when the latter can avoid a function call (or equivalently, its requirement of function calls may make `map` slower).

For comparison, see [Chapter 21](#) for a *nondecorator* approach to timing iteration alternatives like these. As a review, we saw two per-call timing techniques there, homegrown and library—here deployed to time the 1M list comprehension case of the decorator's test code, though incurring extra admin costs that skew results slightly (add [Chapter 21](#)'s folder to your `PYTHONPATH` or `sys.path`, or go there to run this):


```

>>> def listcomp(N): [x * 2 for x in range(N)]

>>> import timer                                     # Chapter 21 techniques
>>> timer.total(1, listcomp, 1_000_000)
(0.08150088600814342, None)
>>> timer.bestoftotal(5, 1, listcomp, 1_000_000)
(0.059792334999656305, None)

>>> import timeit
>>> timeit.timeit(number=1, stmt=lambda: listcomp(1_000_000))
0.08125517799635418
>>> min(timeit.repeat(repeat=5, number=1, stmt=lambda: listcomp(1_000_000)))
0.06156357398140244

```

In this specific case, a nondecorator approach would allow the subject functions to be used with or without timing, but it would also complicate the call signature when timing is desired—we'd need to add code at every call instead of once at the `def`. Moreover, in the nondecorator scheme, there would be no direct way to guarantee that all list builder calls in a program are routed through timer logic, short of finding and potentially changing them all. This may make it difficult to collect cumulative data for all calls.

In general, *decorators* may be preferred when functions are already deployed as part of a larger system and may not be easily passed to analysis functions at calls. On the other hand, because decorators charge each call to a function with augmentation logic, a *nondecorator* approach may be better if you wish to augment calls more selectively. As usual, different tools serve different roles.

Adding Decorator Arguments

The timer decorator of the prior section works, but it would be nice if it were more configurable—providing an output label and turning trace messages on and off, for instance, might be useful in a general-purpose tool like this. Decorator *arguments* come in handy here: when they're coded properly, we can use them to specify configuration options that can vary for each decorated function. A label, for instance, might be added as abstractly follows:

```

def timer(label=''):
    def decorator(func):
        def onCall(*args):
            # Multilevel state retention:
            # args passed to function
            # func retained in enclosing scope
            # label retained in enclosing scope
            func(*args)
            print(label, ...)
        return onCall
    return decorator

    # Returns the actual decorator

@timer('==>')
def listcomp(N): ...

    # Like listcomp = timer('==>')(listcomp)
    # listcomp is rebound to new onCall

listcomp(...)

    # Really calls onCall

```

This code adds an enclosing scope to retain a decorator argument for use on a later actual call. When the `listcomp` function is defined, Python really invokes `decorator`—the result of `timer`, run before decoration actually occurs—with the `label` value available in its enclosing scope. That is, `timer` *returns* the decorator, which remembers both the decorator argument and the original function, and returns the callable `onCall`, which ultimately invokes the original function on later calls. Because this structure creates new `decorator` and `onCall` functions, their enclosing scopes are per-decoration state retention.

We can put this structure to use in our timer to allow a label and a trace control flag to be passed in at decoration time. [Example 39-10](#) does just that, coded in a module file so it can be imported as a general tool; it uses a class for the second state retention level instead of a nested function, but the net result is similar.

Example 39-10. timerdeco2.py

```
import time

def timer(label='', trace=True):                                # On decorator args: retain args
    class Timer:
        def __init__(self, func):                               # On @: retain decorated func
            self.func = func
            self.alltime = 0
        def __call__(self, *args, **kwargs):                     # On calls: call original
            start = time.perf_counter()
            result = self.func(*args, **kwargs)
            elapsed = time.perf_counter() - start
            self.alltime += elapsed
            if trace:
                if label: print(label, end=' ')
                print(f'{self.func.__name__}: {elapsed:.5f}, {self.alltime:.5f}')
            return result
    return Timer
```

Mostly, all we've done here is embed the original `Timer` class in an enclosing function in order to create a scope that retains the decorator arguments per deployment. The outer `timer` function is called before decoration occurs, and it simply returns the `Timer` class to serve as the actual decorator. On decoration, an instance of `Timer` is made that remembers the decorated function itself, but also has access to the decorator arguments in the enclosing function scope.

This time, rather than embedding self-test code in this file, we'll run the decorator in a different file. [Example 39-11](#) is a client of our timer decorator, applying it to sequence iteration alternatives again.

Example 39-11. testseqs.py

```
import sys
from timerdeco2 import timer

@timer(label='[CCC]==>')
def listcomp(N):
    return [x * 2 for x in range(N)]

# Like listcomp = timer(...)(listcomp)
# listcomp(...) triggers Timer.__call__

@timer(trace=True, label='[MMM]==>')
def mapcall(N):
    return list(map((lambda x: x * 2), range(N)))

for func in (listcomp, mapcall):
    result = func(5)
    func(50_000)
    func(500_000)
    func(1_000_000)
    print(result)
    print(f'alltime = {func.alltime}\n')

# Total time for all calls

print('**map/comp =', round(mapcall.alltime / listcomp.alltime, 3))
```

When run, this file prints the following—each decorated function now has a label of its own defined by decorator arguments, which may be more useful when we need to find trace displays mixed in with a larger program’s output:

```
$ python3 testseqs.py
[CCC]==> listcomp: 0.00000, 0.00000
[CCC]==> listcomp: 0.00379, 0.00379
[CCC]==> listcomp: 0.03142, 0.03521
[CCC]==> listcomp: 0.05188, 0.08709
[0, 2, 4, 6, 8]
allTime = 0.08709081003325991

[MMM]==> mapcall: 0.00001, 0.00001
[MMM]==> mapcall: 0.00401, 0.00402
[MMM]==> mapcall: 0.04025, 0.04427
[MMM]==> mapcall: 0.07776, 0.12203
[0, 2, 4, 6, 8]
allTime = 0.12203056103317067

**map/comp = 1.401
```

Run additional tests on your own to see how the decorator’s configuration arguments come into play. As is, this timing function decorator can be used for any function, both in modules and interactively. In other words, it automatically serves as a *general-purpose tool* for timing code in our scripts. Watch for additional examples of decorator arguments ahead when we code decorators to implement attribute privacy and argument range checking.



Timing methods: This section’s timer decorator works on any *function*, but a minor rewrite is required to apply it to class-level *methods* too. In short, and per “[Class Pitfall: Decorating Methods](#)” on page 1048, it must avoid using a nested class. Because this last mutation is being saved for an end-of-chapter quiz question, though, you’ll have to stay tuned for its final code.

Coding Class Decorators

So far, we’ve been coding function decorators to manage function *calls*, but as we’ve seen, decorators work on classes too. As described earlier, while similar in concept to function decorators, class decorators are applied to classes instead—they may be used either to manage *classes* themselves or to intercept instance-creation calls in order to manage *instances*. Also like function decorators, class decorators are really just optional syntactic sugar, though they can make a programmer’s intent more obvious and minimize erroneous or missed calls.

Singleton Classes

Let’s start with something simple. By intercepting instance-creation calls, class decorators can be used to either manage all the instances of a class, or augment the interfaces of those instances. [Example 39-12](#) lists a first class decorator example that does the former—managing all instances of a class. This code implements the classic *singleton* coding pattern, where at most one instance of a class ever exists. Its *singleton* function defines and returns a function for managing instances, and the `@` syntax automatically wraps up a subject class in this function.

Example 39-12. *singletons1.py*

```
instances = {}

def singleton(aClass):
    def onCall(*args, **kwargs):
        if aClass not in instances:
            instances[aClass] = aClass(*args, **kwargs)
        return instances[aClass]
    return onCall
```

To use this, decorate the classes for which you want to enforce a single-instance model, as in [Example 39-13](#).

Example 39-13. *singletons-test.py*

```
from singletons1 import singleton

@singleton
class Person:
    def __init__(self, name, hours, rate):
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate

@singleton
class Hack:
    def __init__(self, val):
        self.attr = val

sue = Person('Sue', 50, 20)
print(sue.name, sue.pay())

bob = Person('Bob', 40, 10)
print(bob.name, bob.pay())

X = Hack(val=42)
Y = Hack(99)
print(X.attr, Y.attr)
```

Now, when the `Person` or `Hack` class is later used to create an instance, the wrapping logic layer provided by the decorator routes instance-creation calls to `onCall`, which in turn ensures a single instance per class, regardless of how many construction calls are made. Here's this code's output when run via command line:

```
$ python3 singletons.py
Sue 1000
Sue 1000
42 42
```

Singleton coding alternatives

Interestingly, you can code a more self-contained solution here with the `nonlocal` statement to change *enclosing-scope* names as described earlier. The following alternative achieves an identical effect, by using one enclosing scope per class, instead of one global table entry per class. It works the same, but it

does not depend on names in the global scope outside the decorator (the `None` check could use `is` instead of `==` here, but it's a trivial test either way):

```
def singleton(aClass):
    instance = None
    def onCall(*args, **kwargs):
        nonlocal instance
        if instance == None:
            instance = aClass(*args, **kwargs)
        return instance
    return onCall
```

On @ decoration
On instance creation
One scope per class

You can also code a self-contained solution with either function attributes or a class instead. The first of the following codes the former, leveraging the fact that there will be one `onCall` function per decoration—the function object's namespace serves the same role as an enclosing scope. The second uses one *instance* per decoration, rather than an enclosing scope, function object, or global table. In fact, the second option relies on the same coding pattern that we will later label a common decorator class pitfall—here we *want* just one instance, but that's not often the case:

```
def singleton(aClass):
    def onCall(*args, **kwargs):
        if onCall.instance == None:
            onCall.instance = aClass(*args, **kwargs)
        return onCall.instance
    onCall.instance = None
    return onCall

class singleton:
    def __init__(self, aClass):
        self.aClass = aClass
        self.instance = None
    def __call__(self, *args, **kwargs):
        if self.instance == None:
            self.instance = self.aClass(*args, **kwargs)
        return self.instance
```

On @ decoration
On instance creation
One function per class
One instance per class

To make this singleton decorator a fully general-purpose tool, choose one version, store it in an importable module file, and indent the self-test code under a `__name__` check—steps we'll leave as suggested exercise. The final class-based version offers an explicit option with extra structure that may better support later evolution, but OOP might not be warranted in all contexts.

Tracing Object Interfaces

The singleton example of the prior section illustrated using class decorators to manage *all* the instances of a class. Another common use case for class decorators augments the interface of *each* generated instance. Class decorators can essentially install a wrapper or “*proxy*” logic layer atop instances that manages access to their interfaces.

For example, in [Chapter 31](#), the `__getattr__` operator-overloading method was shown as a way to wrap up entire object interfaces of embedded instances in order to implement the *delegation* coding pattern. We saw similar examples in the managed attribute coverage of the prior chapter. Recall that `__getattr__` is run when an undefined attribute name is fetched; we can use this hook to intercept method calls in a controller class and propagate them to an embedded object.

The nondecorator approach

For reference and review, here's the original *nondecorator* delegation example:

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object          # Save object
    def __getattr__(self, attrname):
        print('Trace:', attrname)      # Trace fetch
        return getattr(self.wrapped, attrname) # Delegate fetch

x = Wrapper([1,2,3])                  # Wrap a list object
x.append(4)                            # Delegate to list method
```

In this code, the `Wrapper` class intercepts access to any of the wrapped object's explicitly named attributes, prints a trace message, and uses the `getattr` built-in to pass off the request to the wrapped object. Specifically, it traces attribute accesses made *outside* the wrapped object's class; accesses inside the wrapped object's methods are not caught and run normally by design. This *whole-interface* model differs from the behavior of function decorators, which wrap up just one specific method.

The class-decorator approach

Class decorators provide an alternative and convenient way to code this `__getattr__` technique and wrap an entire interface. The preceding code, for example, can be coded as a class decorator that triggers wrapped instance creation instead of passing a premade instance into the wrapper's constructor.

Example 39-14 codes this mod, also supports keyword arguments with `**kargs`, and counts the number of accesses to illustrate changeable state.

Example 39-14. *interfacetracer.py*

```
def Tracer(aClass):                    # On @ decorator
    class Wrapper:
        def __init__(self, *args, **kargs): # On instance creation
            self.fetches = 0
            self.wrapped = aClass(*args, **kargs) # Use enclosing-scope name
        def __getattr__(self, attrname):
            print('Trace: ' + attrname)          # Catches all but own attrs
            self.fetches += 1
            return getattr(self.wrapped, attrname) # Delegate to wrapped obj
    return Wrapper

if __name__ == '__main__':

    @Tracer
    class Hack:
        def display(self):
            print('Hack!' * 3)
        # Hack = Tracer(Hack)
        # Hack is rebound to Wrapper

    @Tracer
    class Person:
        def __init__(self, name, hours, rate): # Person = Tracer(Person)
            self.name = name
            self.hours = hours
            self.rate = rate
        def pay(self):
            return self.hours * self.rate
        # Wrapper remembers Person
        # Accesses outside class traced
        # In-method accesses not traced
```

```

work = Hack()                                # Triggers Wrapper()
work.display()                               # Triggers __getattr__
print([work.fetches])

print()
bob = Person('Bob', 40, 50)                  # bob is really a Wrapper
print(bob.name)                             # Wrapper embeds a Person
print(bob.pay())

print()
sue = Person('Sue', rate=100, hours=60)      # sue is a different Wrapper
print(sue.name)                             # With a different Person
print(sue.pay())

print()
print(bob.name)                             # bob's state != sue's state
print(bob.pay())
print('calls:', [bob.fetches, sue.fetches]) # Wrapper attrs are not traced

```

It's important to note that this is very different from the tracer decorator we met earlier (despite the name!). In [“Coding Function Decorators” on page 1043](#), we looked at decorators that enabled us to trace and time *calls* to a given function or method. In contrast, by intercepting instance-creation calls, the class decorator here allows us to trace an entire object *interface*—that is, accesses to any of the instance's attributes.

It's also important to note that this decorator's `__getattr__` won't catch the implicit attribute fetches of *built-in operations* per the prior chapter, but we'll defer more on this subject until we code attribute privacy ahead.

The following is the output produced by this code: attribute fetches on instances of both the `Hack` and `Person` classes invoke the `__getattr__` logic in the `Wrapper` class because `work`, and `bob`, and `sue` are really instances of `Wrapper`, thanks to the decorator's redirection of instance-creation calls:

```

$ python3 interfacetracer.py
Trace: display
Hack!Hack!Hack!
[1]

Trace: name
Bob
Trace: pay
2000

Trace: name
Sue
Trace: pay
6000

Trace: name
Bob
Trace: pay
2000
calls: [4, 2]

```

Notice how there is one `Wrapper` class with state retention per decoration, generated by the nested class statement in the `Tracer` function, and how each instance gets its own `fetches` counter by virtue of generating a new `Wrapper` instance. As you'll see ahead, orchestrating this is trickier than you may expect.

Applying class decorators to built-in types

Also notice that the preceding decorates a user-defined class. Just like in the original example in [Chapter 31](#), we can also use the decorator to wrap up a built-in object type such as a list, as long as we either subclass to allow decoration syntax or perform the decoration rebinding manually—decorator syntax requires a class statement for the @ line. In the following, x is really a Wrapper again due to the indirection of decoration:

```
>>> from interfacetracer import Tracer

>>> @Tracer
... class MyList(list): pass      # MyList = Tracer(MyList)

>>> x = MyList([1, 2, 3])        # Triggers Wrapper()
>>> x.append(4)                  # Triggers __getattr__, append
Trace: append
>>> x.wrapped
[1, 2, 3, 4]

>>> MyList = Tracer(list)        # Or perform decoration manually
>>> x = MyList([4, 5, 6])        # Else subclass statement required
>>> x.append(7)
Trace: append
>>> x.wrapped
[4, 5, 6, 7]
```

The decorator approach allows us to move instance creation into the decorator itself instead of requiring a premade object to be passed in. Although this seems like a minor difference, it lets us retain normal instance-creation syntax and limits augmentation syntax to class definition. Rather than requiring all instance-creation calls to route objects through a wrapper manually, we need only augment class definitions with decorator syntax:

```
@Tracer                                # Decorator approach
class Person: ...
bob = Person('Bob', 40, 50)
sue = Person('Sue', rate=100, hours=60)

class Person: ...                        # Nondecorator approach
bob = Wrapper(Person('Bob', 40, 50))
sue = Wrapper(Person('Sue', rate=100, hours=60))
```

Assuming you will make more than one instance of a class and want to apply the augmentation to every instance of a class, decorators will generally be a net win in terms of both code size and code maintenance.

Class Pitfall: Retaining Multiple Instances

Curiously, the decorator function in the preceding example can *almost* be coded as a class instead of a function with the proper operator-overloading protocol. [Example 39-15's](#) alternative coding works similarly because its `__init__` is triggered when the @ decorator is applied to the class, and its `__call__` is triggered when a subject class instance is created. Our objects are really instances of Tracer this time, and we essentially just trade an enclosing-scope reference for an instance attribute here.

Example 39-15. *interfacetracer-fail.py* (start)

```
class Tracer:
    def __init__(self, aClass):          # On @decorator
        self.aClass = aClass            # Use instance attribute
    def __call__(self, *args):           # On instance creation
        self.wrapped = self.aClass(*args) # ONE (LAST) INSTANCE PER CLASS!
        return self
    def __getattr__(self, attrname):
        print('Trace:', attrname)
        return getattr(self.wrapped, attrname)

@Tracer                                  # Triggers __init__
class Hack:                              # Like: Hack = Tracer(Hack)
    def display(self):
        print('Hack!' * 3)

work = Hack()                            # Triggers __call__
work.display()                           # Triggers __getattr__
```

As we saw in the abstract earlier, though, this class-only alternative handles multiple *classes* as before, but it won't quite work for multiple *instances* of a given class: each instance-creation call triggers `__call__`, which overwrites the prior instance. The net effect is that Tracer saves just one instance—the last one created. [Example 39-16](#) extends this file to demo the problem.

Example 39-16. *interfacetracer-fail.py* (continued)

```
@Tracer
class Person:                            # Person = Tracer(Person)
    def __init__(self, name):             # Person rebound to a Tracer
        self.name = name

bob = Person('Bob')                      # bob is really a Tracer
print(bob.name)                          # Tracer embeds a Person
sue = Person('Sue')
print(sue.name)                          # sue overwrites bob
print(bob.name)                          # OOPS: now bob's name is 'Sue'!
```

This code's output follows—because this tracer only has a single shared instance, the second overwrites the first:

```
$ python3 interfacetracer-fail.py
Trace: display
Hack!Hack!Hack!
Trace: name
Bob
Trace: name
Sue
Trace: name
Sue
```

The problem here is bad *state retention*—we make one decorator instance per class but not per class instance, such that only the last instance is retained. The solution, as in our prior class pitfall for decorating methods, lies in abandoning class-based decorators.

The earlier function-based Tracer version of [Example 39-14](#), however, *does* work for multiple instances. Because it returns a *class* instead of an *instance* of that class, each instance-creation call makes a

new `Wrapper` instance instead of overwriting the state of a single shared `Tracer` instance. The original nondecorator version handles multiple instances correctly for the same reason. The moral here: decorators are not only arguably magical, they can also be incredibly subtle!

Example: “Private” and “Public” Attributes

The final two sections of this chapter present larger examples of decorator use, which give us a chance to see how concepts come together in more useful code. Both are presented with minimal description, partly to conserve space but mostly because you should already understand decorator basics well enough to be able to study these on your own.

Implementing Private Attributes

First up, the *class decorator* in [Example 39-17](#) implements a `Private` declaration and access checks for class instance attributes—that is, for attributes stored on an instance, or inherited from one of its classes.

This decorator disallows fetch and change access to such attributes from *outside* the decorated class but still allows the class itself to access those names freely within its own methods. It’s not quite the same as “private” in C++ or Java—and Python is not about control in general—but this decorator demo provides similar access validations as an option in Python for the rare and atypical cases where this might be useful during development.

We saw an initial and incomplete implementation of instance attribute privacy for *changes* only in [Chapter 30](#). The version here extends this concept to validate attribute *fetches*, too, and it uses delegation instead of inheritance to implement the model. In a sense, this is also just an extension to the attribute-tracer class decorator we met earlier.

Although this example utilizes the syntactic sugar of class decorators to code attribute privacy, its attribute interception is ultimately still based upon the `__getattr__` and `__setattr__` operator-overloading methods we met in prior chapters. When a private attribute access is detected, this version uses the `raise` statement to raise an exception, along with an error message; the exception may be caught in a `try` or allowed to terminate the accessing script.

[Example 39-17](#) lists the decorator’s first-cut code, along with a self-test at the bottom of the file. As coded, it catches all explicit attribute fetches, but not the implicit fetches of built-in operations (more on this in a moment).

Example 39-17. access1.py

```
"""
Class decorator with Private attribute declarations.

Privacy for attributes fetched from class instances.
See self-test code at end of file for a usage example.

Rebinding is: Doubler = Private('data', 'size')(Doubler).
Private returns onDecorator, onDecorator returns onInstance,
and each onInstance instance embeds a new Doubler instance.
"""

traceMe = False
def trace(*args):
    if traceMe: print(f'[{ ' '.join(map(str, args))}]') # Python 3.12+ f-string
```

```

def Private(*privates):
    # privates in enclosing scope
    def onDecorator(aClass):
        # aClass in enclosing scope
        class onInstance:
            # wrapped in instance attribute
            def __init__(self, *args, **kwargs):
                self.wrapped = aClass(*args, **kwargs)

            def __getattr__(self, attr):
                # My attrs don't call getattr
                trace('get:', attr)
                # Others assumed in wrapped
                if attr in privates:
                    raise TypeError('private attribute fetch, ' + attr)
                else:
                    return getattr(self.wrapped, attr)

            def __setattr__(self, attr, value):
                # Outside accesses
                trace('set:', attr, value)
                # Others run normally
                if attr == 'wrapped':
                    # Allow my attrs
                    self.__dict__[attr] = value
                    # Avoid looping
                elif attr in privates:
                    raise TypeError('private attribute change, ' + attr)
                else:
                    setattr(self.wrapped, attr, value)
                    # Wrapped obj attrs
        return onInstance
    return onDecorator

if __name__ == '__main__':
    traceMe = True

    @Private('data', 'size')
    # Doubler = Private(...)(Doubler)
    class Doubler:
        def __init__(self, label, start):
            self.label = label
            self.data = start
            # Accesses inside the subject class
            # Not intercepted: run normally
        def size(self):
            return len(self.data)
            # Method bodies run with no checking
        def double(self):
            # Because privacy not inherited
            for i in range(self.size()):
                self.data[i] = self.data[i] * 2
        def display(self):
            print(f'{self.label} => {self.data}')

    print('Making instances...')
    X = Doubler('X is', [1, 2, 3])
    Y = Doubler('Y is', [-10, -20, -30])

    # The following all succeed properly

    print('\nExploring X instance...')
    print(X.label)
    X.display(); X.double(); X.display()
    # Accesses outside subject class
    # Intercepted: validated, delegated

    print('\nExploring Y instance...')
    print(Y.label)
    Y.display(); Y.double()
    Y.label = 'Hack'
    Y.display()

    # The following all fail properly
    """

```

```

print(X.size())          # Prints "TypeError: private attribute fetch, size"
print(X.data)
X.data = [1, 1, 1]       # Prints "TypeError: private attribute change, data"
X.size = lambda S: 0
print(Y.data)
print(Y.size())
"""

```

When its `traceMe` is `True`, the module file's self-test code produces the following output. Notice how the decorator catches and validates both attribute fetches and assignments run *outside* of the wrapped class but does not catch attribute accesses *inside* the class itself:

```

$ python3 access1.py
Making instances...
[set: wrapped <__main__.Doubler object at 0x1059c7d70>]
[set: wrapped <__main__.Doubler object at 0x1059c7da0>]

Exploring X instance...
[get: label]
X is
[get: display]
X is => [1, 2, 3]
[get: double]
[get: display]
X is => [2, 4, 6]

Exploring Y instance...
[get: label]
Y is
[get: display]
Y is => [-10, -20, -30]
[get: double]
[set: label Hack]
[get: display]
Hack => [-20, -40, -60]

```

Implementation Details I

This code is nontrivial, and you're probably best off tracing through it on your own to see how it works. To help you study, though, here are a few highlights worth mentioning.

Inheritance versus delegation

The initial and limited privacy example shown in [Chapter 30](#) used *inheritance* to mix in a `__setattr__` to catch accesses. Inheritance makes this difficult, however, because differentiating between accesses from inside or outside the class is not straightforward (inside access should be allowed to run normally, and outside access should be restricted). To work around this, the [Chapter 30](#) example requires inheriting classes to use `__dict__` assignments to set attributes—an incomplete solution at best.

The version here uses *delegation* (embedding one object inside another) instead of inheritance; this pattern is better suited to our task as it makes it much easier to distinguish between accesses inside and outside of the subject class. Attribute accesses from outside the subject class are intercepted by the wrapper layer's overloading methods and delegated to the class if valid. Accesses inside the class itself (i.e., through `self` within its methods' code) are not intercepted and are allowed to run normally without checks because privacy is not inherited in this version.

Decorator arguments

The class decorator used here accepts any number of arguments to name private attributes. Again, though, this simply means that the arguments are passed to the `Private` function, and `Private` returns the decorator function to be applied to the subject class. That is, the arguments are used before decoration ever occurs; `Private` returns the decorator, which in turn “remembers” the `privates` list as an enclosing-scope reference.

State retention and enclosing scopes

Speaking of enclosing scopes, there are actually *three levels* of state retention at work in this code:

- The arguments to `Private` are used before decoration occurs and are retained as an enclosing-scope reference for use in both `onDecorator` and `onInstance`.
- The class argument to `onDecorator` is used at decoration time and is retained as an enclosing-scope reference for use at instance-creation time.
- The wrapped instance object is retained as an instance attribute in the `onInstance` proxy object for use when attributes are later accessed from outside the class.

This all works fairly naturally, given Python’s scope and namespace rules.

Using `__dict__` and `__slots__` (and other virtuals)

The `__setattr__` method in this code relies on an instance object’s `__dict__` attribute namespace dictionary in order to set `onInstance`’s own wrapped attribute. As we learned in the prior chapter, this method cannot assign an attribute directly without looping. However, it uses the `setattr` built-in instead of `__dict__` to set attributes in the *wrapped* object itself. Moreover, `getattr` is used to fetch attributes in the *wrapped* object since they may be stored in the object itself or inherited by it.

Because of that, this code will work for most classes—including those with “virtual” class-level attributes based on *slots*, *properties*, *descriptors*, and even `__getattr__` and its ilk. By assuming a namespace dictionary for itself only and using storage-neutral tools for the wrapped object, the wrapper class avoids limitations imposed by other tools.

For example, you may recall from [Chapter 32](#) that classes with `__slots__` may not store attributes in a `__dict__`, and in fact, may not even have one of these at all. However, because we rely on a `__dict__` only at the `onInstance` level here and not in the wrapped instance, this concern does not apply. Class `onInstance` will have a `__dict__` itself because it does not use slots. In addition, because `setattr` and `getattr` apply to attributes based on both `__dict__` and `__slots__`, our decorator applies to wrapped classes using either storage scheme.

By the same reasoning, the decorator also applies to properties and similar tools: delegated names will be looked up anew in the wrapped instance, irrespective of attributes of the decorator proxy object itself.

Generalizing for Public Declarations

Now that we have a `Private` attribute implementation, it’s straightforward to generalize the code to allow for `Public` declarations too—they are essentially the inverse of `Private` declarations, so we need only negate the inner test. The example listed in [Example 39-18](#) allows a class to use decorators to

define a set of either `Private` or `Public` instance attributes—attributes of any kind stored on an instance or inherited from its classes—with the following semantics:

- `Private` declares attributes of a class's instances that *cannot* be fetched or assigned except from within the code of the class's methods. That is, any name declared `Private` cannot be accessed from outside the class, while any name not declared `Private` can be freely fetched or assigned from outside the class.
- `Public` declares attributes of a class's instances that *can* be fetched or assigned from both outside the class and within the class's methods. That is, any name declared `Public` can be freely accessed anywhere, while any name not declared `Public` cannot be accessed from outside the class.

`Private` and `Public` declarations are mutually exclusive: when using `Private`, all undeclared names are considered `Public`, and when using `Public`, all undeclared names are considered `Private`. They are essentially opposites, though undeclared names not created by a class's methods behave slightly differently—new names can be assigned and thus created outside the class under `Private` (all undeclared names are accessible) but not under `Public` (all undeclared names are inaccessible).

Again, study this code on your own to get a feel for how this works. Notice that this scheme adds an additional *fourth level of state retention* at the top, beyond that described in the preceding section: the validation functions used by the lambdas are saved in an extra enclosing scope coded separately. This version comes with the same caveat as its predecessor for attributes of built-in operations, noted in the file's docstring and expanded on after the example.

Example 39-18. *access2.py*

```
"""
Class decorator with Private and Public attribute declarations.

Controls external access to attributes stored on an instance, or
inherited by it from its classes. Private declares attribute names
that cannot be fetched or assigned outside the decorated class,
and Public declares all the names that can. Choose either decorator.

Caveat: as is, this works for explicitly-named attributes only. The
__X__ operator-overloading methods fetched implicitly for built-in
operations do not trigger either __getattr__ or __getattribute__, and
hence won't be delegated to any wrapped objects that define them. If
needed, add __X__ methods to catch and delegate built-ins (per ahead).
"""

traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)

            def __getattr__(self, attr):
                trace('get:', attr)
                if failIf(attr):
                    raise TypeError('private attribute fetch, ' + attr)
                else:
                    return getattr(self.__wrapped, attr)
```

```

    def __setattr__(self, attr, value):
        trace('set:', attr, value)
        if attr == '_onInstance_wrapped':
            self.__dict__[attr] = value
        elif failIf(attr):
            raise TypeError('private attribute change, ' + attr)
        else:
            setattr(self.__wrapped, attr, value)
    return onInstance
return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))

def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))

```

See the prior example's self-test code for a usage example—the effect is the same for `Private`. Here's a quick look at these class decorators in action at the interactive prompt. As advertised, non-`Private` or `Public` names can be fetched and changed from outside the subject class, but `Private` or non-`Public` names cannot:

```

>>> from access2 import Private, Public

>>> @Private('age')
... class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
# Person = Private('age')(Person)
# Person = onInstance with state
# Inside accesses run normally

>>> X = Person('Pat', 40)
>>> X.name
'Pat'
# Outside accesses validated
>>> X.name = 'Sue'
>>> X.name
'Sue'
>>> X.age
TypeError: private attribute fetch, age
>>> X.age = 'Bob'
TypeError: private attribute change, age

>>> @Public('name')
... class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

>>> X = Person('Pat', 40)
>>> X.name
'Pat'
# X is an onInstance
# onInstance embeds Person
>>> X.name = 'Sue'
>>> X.name
'Sue'
>>> X.age
TypeError: private attribute fetch, age
>>> X.age = 'Bob'
TypeError: private attribute change, age

```

Implementation Details II

To help you analyze [Example 39-18](#)’s code, here are a few final notes on this version. Since this is just a generalization of the preceding section’s version, the implementation notes there apply here as well.

Using “__X” pseudoprivate names

Besides generalizing, this version also makes use of Python’s `__X` pseudoprivate name mangling feature, which we met in [Chapter 31](#), to localize the wrapped attribute to the proxy control class by automatically prefixing it with this class’s name. This avoids the prior version’s risk for collisions with a wrapped attribute that may be used by the real, wrapped class, and it’s useful in a general tool like this. It’s not quite “privacy,” though, because the mangled version of the name can be used freely outside the class. Notice that we also have to use the fully expanded name string—`'_onInstance__wrapped'`—as an admin-name test value in `__setattr__` because that’s what Python changes it to.

Breaking privacy

Although this example does implement access controls for attributes of an instance and its classes, it is possible to subvert these controls trivially—for instance, by fetching through the expanded version of the wrapped attribute explicitly (`bob.pay` might not work, but the fully mangled `bob._onInstance__wrapped.pay` could!). If you have to try that hard to break them, though, these tools probably suffice for intended roles. Of course, privacy can generally be subverted in other languages too (e.g., `#define private public` may work in some C++ implementations). Although access controls may reduce accidental mods, much of this is up to programmers in any language; whenever source code may be changed, airtight access control will always be a pipe dream. More fundamentally, Python is about *enabling*, not controlling; privacy is a tool best used sparingly (if at all).

Decorator trade-offs

We could again achieve the same results without decorators by using helper functions or coding the name rebinding of decorators manually; the decorator syntax, however, makes this consistent and obvious in code. The chief potential downsides of this and any other wrapper-based approach are that attribute access incurs an extra call, and instances of decorated classes are not really instances of the original decorated class—if you test their type with `X.__class__` or `isinstance(X, C)`, for example, you’ll find that they are instances of the *wrapper* class. Unless you plan to do introspection on objects’ types, though, the type issue is irrelevant, and the extra call may apply mostly to development time; as you’ll see later, it’s possible to remove decorations automatically (via `-O`) if desired.

Delegating Built-In Operations

As is, this section’s examples work as planned for methods and other attributes fetched *explicitly* by name. As with most software, though, there is always room for improvement. Most notably, this tool turns in mixed performance on operator-overloading methods if they are used by client classes.

Specifically, the proxy class fails to validate or delegate operator-overloading methods fetched *implicitly* by built-in operations unless such methods are redefined in the proxy. Clients that do not use operator overloading are fully supported, but others may require additional code. It’s unclear that operator-overloading methods *should* be validated as private or public, but they are a part of an object’s interface and should at least be routed to wrapped objects that define them.

We’ve encountered this issue a few times already in this book, but let’s take a quick look at its impact on the realistic code we’ve written here and explore workarounds for it. The basic issue is easy to demo—as we’ve learned, the following is how a class that overloads `print` calls and `+` expressions normally works:

```
>>> class Tally:
    def __init__(self):
        self.sum = 0
    def __str__(self):
        return f'Tally: {self.sum}'
    def __add__(self, add):
        self.sum += add

>>> X = Tally()
>>> X.sum           # All attributes accessible
0
>>> print(X)        # Same as X.__str__() {sort of}
Tally: 0
>>> X + 5           # Same as X.__add__(5) {ditto}
>>> print(X)
Tally: 5
```

Unfortunately, objects that implement built-in operations like this fail in our proxy classes because built-in operations *skip* instance-level lookup protocols like `__getattr__`, and instead search namespaces of classes:

```
>>> from access2 import Private
>>> @Private('sum', '__add__')
... class Tally:
    ...same as before...

>>> X = Tally()
>>> X.sum
TypeError: private attribute fetch, sum

>>> X.__add__(5)
TypeError: private attribute fetch, __add__

>>> print(X)
<access2.accessControl.<locals>.onDecorator.<locals>.onInstance object at 0x...etc...>

>>> X + 5
TypeError: unsupported operand type(s) for +: 'onInstance' and 'int'
```

In this session, the first two *explicit* fetches of `sum` and `__add__` are kicked out as *privates* as they *should* be. Because the last two *implicit* fetches of `print` and `+` aren’t caught by the proxy, though, they are never delegated to the wrapped `Tally` object. The `print` here only works at all because it runs an object default to print the proxy itself. Per the prior chapter, this is an inconsistency in Python; per the following sections, it can also be avoided in full.

Workaround: Coding operator-overloading methods inline

The most straightforward way to support built-ins in delegation proxies is to redefine operator-overloading names that may appear in embedded objects. This creates some code redundancy, but it isn’t impossibly onerous; can be automated with tools or superclasses; and can choose to run or skip validations for operator-overloading names declared `Private` or `Public`, depending on redefinitions’ routing.

For instance, the partial listing of [Example 39-19](#) sketches an *inline* redefinition approach—it catches and delegates built-ins by adding method definitions to the proxy itself for every operator-overloading method a wrapped object may define. It adds just four operation interceptors to illustrate, but others are similar (in this section, new code is in bold font, and all examples are based on the decorator of `access2.py` in [Example 39-18](#)).

Example 39-19. `access_builtins_inline_direct.py`

```
"Inline methods, skip validations"

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kwargs):
                self.__wrapped = aClass(*args, **kwargs)

            # Intercept and delegate built-in implicit access specifically

            def __add__(self, other):
                return self.__wrapped + other          # Or getattr(), __getattr__()
            def __str__(self):
                return str(self.__wrapped)              # Or self.__wrapped.__str__()
            def __getitem__(self, index):
                return self.__wrapped[index]
            def __call__(self, *args, **kwargs):
                return self.__wrapped(*args, **kwargs)
            # Plus any others needed

            # Intercept and delegate explicit attribute access generically

            def __getattr__(self, attr): ...same...
            def __setattr__(self, attr, value): ...same...
        return onInstance
    return onDecorator
```

This works because built-ins will find their requisite methods in the proxy *class* after skipping the proxy instance. As coded, the new interceptor methods trigger the wrapped object's operator-overloading methods *directly* and so bypass the access controls of `__getattr__`, which may or may not be desirable. For alternative codings, let's move on.

Workaround: Coding operator-overloading methods in superclasses

More usefully, the prior section's added methods can be added by a common *superclass*. Given that there are dozens of such methods, an external class may be better suited to the task, especially if it is general enough to be used in any such interface-proxy class.

To demo, the superclass of [Example 39-20](#) catches built-ins and reroutes to the wrapped object *directly* again. It's largely just a repackaging of the prior section's inline scheme, but as a separate class it requires a proxy attribute named `_wrapped`, giving access to the embedded object. The decorator itself must use this name instead of `__wrapped` in self references, and sans mangling in `__setattr__`. This may be subpar because it precludes the same name in wrapped objects and creates a subclass dependency, but it's better than using the mangled and subclass-specific `_onInstance__wrapped` and is no worse than a similarly named method.

Example 39-20. *access_builtins_mixin_direct.py*

"Inherit methods, skip validations"

```
class BuiltinsMixin:
    def __add__(self, other):
        return self._wrapped + other                # Assume a _wrapped
    def __str__(self):
        return str(self._wrapped)                   # Bypass __getattr__
    def __getitem__(self, index):
        return self._wrapped[index]
    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)
    # Plus any others needed

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):
            ...rest same, but use unmangled _wrapped instead of __wrapped...
```

Alternatively, the superclass in [Example 39-21](#) catches built-ins and reroutes them down through the subclass `__getattr__` to apply its access controls to the operation's method name. It requires that operator-overloading names be non-Private or Public per the decorator's arguments if they are to be run, but it treats the implicit fetches of built-in operations the same as explicit-name fetches, and no `_wrapped` is required in subclasses.

Example 39-21. *access_builtins_mixin_getattr.py*

"Inherit methods, run validations"

```
class BuiltinsMixin:
    def __add__(self, other):
        return self.__getattr__('__add__')(other)    # Route to validator
    def __str__(self):
        return self.__getattr__('__str__')()         # Finish operations
    def __getitem__(self, index):
        return self.__getattr__('__getitem__')(index)
    def __call__(self, *args, **kwargs):
        return self.__getattr__('__call__')(*args, **kwargs)
    # Plus any others needed

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):
            # Inherit methods
            ...rest unchanged...
```

Like the inline approach, both of these mix-ins also require one method per built-in operation in general tools that proxy arbitrary objects' interfaces. The next idea does marginally better.

Workaround: Generating operator-overloading descriptors

Finally, all of the inline and mix-in workarounds for built-ins we've seen so far code each operator-overloading method explicitly, and intercept the actual *call* issued for the operation, including its arguments. That makes them responsible for completing the operation, whether by operation syntax or equivalent calls.

With an alternative coding, we could instead intercept only the attribute *fetch* preceding the call by using the class-level *descriptors* of the prior chapter. Moreover, because all such descriptors will run the same, they can be generated automatically from a list of method names. [Example 39-22](#) shows one way to code this scheme. Like [Example 39-21](#), it routes built-in operations through the decorator’s validations logic to *apply* private or public checks.

Example 39-22. *access_builtins_mixin_desc.py*

```
"Inherit descriptors, run validations"

class BuiltinsMixin:
    class ProxyDesc:
        # Define descriptor
        def __init__(self, attrname):
            self.attrname = attrname
        def __get__(self, instance, owner):
            return instance.__getattr__(self.attrname)
            # Run validations

    builtins = ['add', 'str', 'getitem', 'call']
    # Plus any others
    for attr in builtins:
        exec(f'__{attr}__ = ProxyDesc("__{attr}__")')
        # Make descriptors

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):
            # Inherit descriptors
            ...rest unchanged...
```

This coding may be the most concise but also the most implicit and complex. Recall that the `exec` built-in by default runs a string of code as if the string was somehow pasted where the `exec` appears. Hence, the loop at the end of this mix-in class is equivalent to the following statements, run in the mix-in class’s local scope:

```
__add__ = ProxyDesc("__add__")
__str__ = ProxyDesc("__str__")
...etc...
```

The net effect creates inherited descriptor instances that respond to initial name lookups by fetching from the wrapped object in `__get__` rather than catching the later operation call itself (which happens after this step). If you still find this code confusing (and you probably should), it’s equivalent to this stripped-down version, though the name fetch occurs implicitly in a built-in operation that skips the instance’s protocols:

```
>>> class B:
    class D:
        def __get__(s, i, o): return i.meth()
        name = D()

>>> class A(B):
    def meth(self): return 'hack'

>>> I = A()
>>> I.name
'hack'
```

We could also *skip* the decorator’s validations for built-in operations in this scheme by routing attribute fetches directly to the wrapped object—though this requires an accessible `_wrapped` in the decorator just like [Example 39-20](#):

```

class ProxyDesc:
    ...
    def __get__(self, instance, owner):
        return getattr(instance._wrapped, self.attrname)    # Assume a _wrapped

```

In the end, all of these workarounds make classes that overload built-in operations work correctly with our private and public decorators—and other delegation-based decorators like them:

```

>>> from access_builtins_mixin_desc import Private

>>> @Private('sum')
... class Tally:
    def __init__(self):
        self.sum = 0
    def __str__(self):
        return f'Tally: {self.sum}'
    def __add__(self, add):
        self.sum += add

>>> X = Tally()
>>> X.sum                                # Explicit validated
TypeError: private attribute fetch, sum
>>> X + 10                               # Built-in delegated
>>> print(X)                             # Built-in delegated
Tally: 10

```

Public (nonprivate) built-ins are now delegated and work, but private built-ins are validated and canceled:

```

>>> @Private('sum', '__add__')
... class Tally:
    ...same as before...

>>> X = Tally()
>>> X.sum                                # Explicit validated
TypeError: private attribute fetch, sum
>>> X + 10                               # Built-in canceled: private
TypeError: private attribute fetch, __add__
>>> print(X)                             # Built-in allowed: public
Tally: 0

>>> @Private('__str__')
... class Tally:
    ...same as before...

>>> print(Tally())                       # Built-in canceled: private
TypeError: private attribute fetch, __str__

```

If you care to experiment further with this section's examples, see the book examples package for their complete code, as well as its comprehensive *access_builtins_TEST.py* test script and results. Here, it's time to move on to this chapter's next and final decorators case study.

Example: Validating Function Arguments

As a final example of the utility of decorators, this section develops a *function decorator* that automatically tests whether arguments passed to a function or method are within a valid numeric range. It's designed to be used during either development or production, and it can be used as a template for similar tasks (e.g., argument type testing, if you must). Again, this example is largely self-study content with a limited narrative; read the code for more details.

The Goal

In the object-oriented tutorial of [Chapter 28](#), we wrote a class that gave a pay raise to objects representing fictitious people, based upon a passed-in percentage:

```
class Person:
    ...
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

There, we noted that if we wanted the code to be robust, it would be a good idea to check the percentage to make sure it's not too large or too small. We could implement such a check with either `if` or `assert` statements in the method itself, using *inline tests*:

```
class Person:
    def giveRaise(self, percent):
        if percent < 0.0 or percent > 1.0:
            raise TypeError, 'percent invalid'
        self.pay = int(self.pay * (1 + percent))

class Person:
    def giveRaise(self, percent):
        assert percent >= 0.0 and percent <= 1.0, 'percent invalid'
        self.pay = int(self.pay * (1 + percent))
```

However, this approach clutters the method with inline tests that will probably be useful only during development. For more complex cases, this can become tedious (imagine trying to inline the code needed to implement the attribute privacy provided by the last section's decorator). Perhaps worse, if the validation logic ever needs to change, there may be arbitrarily many inline copies to find and update.

A more useful and interesting alternative would be to develop a general tool that can perform range tests for us automatically for the arguments of any function or method we might code now or in the future. A *decorator* approach makes this explicit and convenient, and easy to disable once development is complete:

```
class Person:
    @rangetest(percent=(0.0, 1.0))
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

Isolating validation logic in a decorator simplifies both clients and future maintenance.

Notice that our goal here is different than the attribute validations coded in the prior chapter's final example. Here, we mean to validate the values of *function arguments* when passed rather than *attribute values* when accessed. Python's decorator and introspection tools allow us to code this new task just as easily.

A Basic Range-Testing Decorator for Positional Arguments

Let's start with a basic range-test implementation. To keep things simple, we'll begin by coding a decorator that works only for *positional* arguments and assumes they always appear at the same position in every call; they cannot be passed by keyword name because this can invalidate the positions declared in the decorator. [Example 39-23](#) is our first-cut checker.

Example 39-23. rangetest0.py

```
def rangetest(*argchecks):                # Validate positional arg ranges
    def onDecorator(func):
        if not __debug__:                 # True if "python -O main.py args..."
            return func                   # No-op: call original directly
        else:                             # Else wrapper while debugging
            def onCall(*args):
                for (ix, low, high) in argchecks:
                    if args[ix] < low or args[ix] > high:
                        errmsg = f'Argument {ix} not in {low}..{high}'
                        raise TypeError(errmsg)
                return func(*args)
            return onCall
    return onDecorator
```

As is, this code is mostly a rehash of the coding patterns we explored earlier: we use decorator *arguments*, nested *scopes* for state retention, and so on.

We also use nested `def` statements to ensure that this works for both simple functions and *methods*, as we learned earlier. When used for a class's method, `onCall` receives the subject class's instance in the first item in `*args` and passes this along to `self` in the original method function; explicitly passed argument numbers coded in the `@` decorator line start at 1 in this case, not 0, to accommodate the implicit `self`.

New here, notice this code's use of the `__debug__` built-in variable introduced in [Chapter 34](#). In brief, Python sets this variable to `True` unless the program is being run with the `-O` optimize command-line flag (e.g., `python -O main.py`). As discussed earlier, using options in the `compile` built-in function and `compileall` standard-library module before code is run can have a similar effect.

Either way, when `__debug__` is `False`, the decorator returns the original function unchanged to avoid extra later calls and their associated performance penalty. In other words, the decorator automatically *removes* its augmentation logic when `-O` or similar is used without requiring you to physically remove the decoration lines in your code.

Example 39-24 demos how this first-iteration solution is used.

Example 39-24. rangetest0_test.py

```
from rangetest0 import rangetest
print(f'__debug__={__debug__}')          # False if "python -O main.py"

@rangetest((1, 0, 120))                  # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):                  # age must be in 0..120
    print(f'{name} is {age} years old')

@rangetest([0, 1, 12], [1, 1, 31], [2, 0, 2024])
def birthday(M, D, Y):
    print(f'birthday = {M}/{D}/{Y}')

class Person:
    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay

    @rangetest([1, 0.0, 1.0])              # giveRaise = rangetest(...)(giveRaise)
    def giveRaise(self, percent):          # Arg 0 is the self instance here
        self.pay = int(self.pay * (1 + percent))
```

```

# Comment lines raise TypeError unless "python -O" used on shell command line

persinfo('Bob Smith', 45)           # Really runs onCall(...) with state
#persinfo('Bob Smith', 200)        # Or persinfo if -O cmd line argument

birthday(8, 31, 2024)
#birthday(8, 32, 2024)

sue = Person('Sue Jones', 'dev', 100_000)
sue.giveRaise(.10)                  # Really runs onCall(self, .10)
print(sue.pay)                     # Or giveRaise(self, .10) if -O
#sue.giveRaise(1.10)

```

When run, valid calls in this code produce the following output:

```

$ python3 rangetest0_test.py
__debug__=True
Bob Smith is 45 years old
birthday = 8/31/2024
110000

```

Uncommenting any of the invalid calls causes a `TypeError` to be raised by the decorator. Here's the result when the last line is allowed to run (as usual, some of the error message text was trimmed here to save space):

```

$ python3 rangetest0_test.py
__debug__=True
Bob Smith is 45 years old
birthday = 8/31/2024
110000
TypeError: Argument 1 not in 0.0..1.0

```

Running Python with its `-O` flag at a system command line will disable range testing but also avoid the performance overhead of the wrapping layer—we wind up calling the original undecorated function directly. Assuming this is a debugging tool only, you can use this flag to optimize your program for production use. Here is the effect with the last line still run and a print added to show sue's fantastical pay raise:

```

$ python3 -O rangetest0_test.py
__debug__=False
Bob Smith is 45 years old
birthday = 8/31/2024
110000
231000

```

Generalizing for Keywords and Defaults

The prior version illustrates the basics we need to employ, but it's fairly limited—it supports validating arguments passed by position only, and it does not validate keyword arguments (in fact, it assumes that no keywords are passed in a way that makes argument position numbers incorrect). Additionally, it does nothing about arguments with defaults that may be omitted in a given call. That's fine if all your arguments are passed by position and never defaulted, but it's less than ideal in a general tool. As we learned in [Chapter 18](#), Python supports much more flexible argument-passing modes, which we're not yet addressing.

The level up of our decorator in [Example 39-25](#) does better. By matching the wrapped function's expected arguments against the actual arguments passed in a call, it supports range validations for

arguments passed by either position or keyword name, and it skips testing for default arguments omitted in the call. Arguments to be validated are specified by keyword arguments to the decorator itself, which later steps through both the call's *pargs positionals tuple and its **kargs keywords dictionary to validate.

Example 39-25. rangetest.py

```
"""
```

A function decorator that performs range-test validation for arguments passed to any function or method. Usage synopsis:

```
@rangetest(percent=(0.0, 1.0), month=(1, 12))
def func-or-method(..., percent, ..., month=5, ...):
    ...
func-or-method(..., value, month=8, ...)
```

Arguments are specified by keyword to the decorator. In the actual call, arguments may be passed by position or keyword, and defaults may be omitted. See rangetest_test.py for example use cases.

```
"""
```

```
trace = True
```

```
def rangetest(**argchecks):
    # Validate ranges for both+defaults
    def onDecorator(func):
        # onCall remembers func and argchecks
        if not __debug__:
            # True if "python -O main.py args..."
            return func
        # Wrap if debugging; else use original
        else:
            funcname = func.__name__
            funccode = func.__code__
            funcargs = funccode.co_varnames[:funccode.co_argcount]

            def onCall(*pargs, **kargs):
                # All pargs match first N expected args by position
                # The rest must be in kargs or be omitted defaults
                positionals = funcargs[:len(pargs)]
                errormsg = lambda *args: '%s argument "%s" not in %s..%s' % args

                for (argname, (low, high)) in argchecks.items():
                    # For all args to be checked
                    if argname in kargs:
                        # Was passed by name
                        if kargs[argname] < low or kargs[argname] > high:
                            raise TypeError(errormsg(funcname, argname, low, high))

                    elif argname in positionals:
                        # Was passed by position
                        position = positionals.index(argname)
                        if pargs[position] < low or pargs[position] > high:
                            raise TypeError(errormsg(funcname, argname, low, high))

                    else:
                        # Assume not passed: default
                        if trace:
                            print(f'-Argument "{argname}" defaulted')

                return func(*pargs, **kargs) # OK: run original call
            return onCall
    return onDecorator
```

Next, the test script in [Example 39-26](#) shows how the decorator is used—arguments to be validated are given by keyword decorator arguments, and at actual calls, we can pass by name or position and omit arguments with defaults even if they are to be validated otherwise.

Example 39-26. *rangetest_test.py*

```
"""
Test the rangetest decorator (usage differs from rangetest0).
Comment lines raise TypeError unless "python -0" or similar in compileall.
"""

from rangetest import rangetest

def announce(what): print(what.center(24, '-'))    # str method

# Test functions, positional and keyword
announce('Functions')

@rangetest(age=(0, 120))                        # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):
    print(f'{name} is {age} years old')

@rangetest(M=(1, 12), D=(1, 31), Y=(0, 2024))
def birthday(M, D, Y):
    print(f'birthday = {M}/{D}/{Y}')

persinfo('Pat', 40)
persinfo(age=40, name='Pat')
birthday(8, D=31, Y=2024)
#persinfo('Pat', 150)
#persinfo(age=150, name='Pat')
#birthday(8, Y=2025, D=40)

# Test methods, positional and keyword
announce('Methods')

class Person:
    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay

    # giveRaise = rangetest(...)(giveRaise)
    @rangetest(percent=(0.0, 1.0))             # percent passed by name or position
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

sue = Person('Sue Jones', 'dev', 100_000)
bob = Person('Bob Smith', 'dev', 100_000)
sue.giveRaise(percent=.20)
bob.giveRaise(.10)
print(f'sue=>{sue.pay}, bob=>{bob.pay}')
#sue.giveRaise(1.20)
#bob.giveRaise(percent=1.20)

# Test omitted defaults: skipped
announce('Defaults')

@rangetest(a=(1, 10), b=(1, 10), c=(1, 10), d=(1, 10))
def omitargs(a, b=7, c=8, d=9):
    print(a, b, c, d)

omitargs(1, 2, 3, 4)                # Positionals
```

```

omitargs(1, 2, 3)           # Default d
omitargs(1, 2, 3, d=4)      # Keyword d
omitargs(1, d=4)            # Default b and c
omitargs(d=4, a=1)          # Ditto
omitargs(1, b=2, d=4)        # Default c
omitargs(d=8, c=7, a=1)      # Default b

#omitargs(1, 2, 3, 11)       # Bad d
#omitargs(1, 2, 11)          # Bad c
#omitargs(1, 2, 3, d=11)     # Bad d
#omitargs(11, d=4)           # Bad a
#omitargs(d=4, a=11)         # Bad a
#omitargs(1, b=11, d=4)      # Bad b
#omitargs(d=8, c=7, a=11)    # Bad a

```

When this script is run, out-of-range arguments raise an exception as before, but arguments may be passed by either name or position, and omitted defaults are not validated. Trace its output and test this further on your own to experiment; it works like its simpler predecessor, but its scope has been greatly broadened:

```

$ python3 rangetest_test.py
-----Functions-----
Pat is 40 years old
Pat is 40 years old
birthday = 8/31/2024
-----Methods-----
sue=>120000, bob=>110000
-----Defaults-----
1 2 3 4
-Argument "d" defaulted
1 2 3 9
1 2 3 4
-Argument "b" defaulted
-Argument "c" defaulted
1 7 8 4
-Argument "b" defaulted
-Argument "c" defaulted
1 7 8 4
-Argument "c" defaulted
1 2 8 4
-Argument "b" defaulted
1 7 7 8

```

Notice that argument checks are run in the *order* they are listed in the decorator because Python retains insertion order in dictionaries. On validation errors, we get an exception as before unless the `-0` command-line argument is passed to Python to disable the decorator's logic. Here's the scene when one of the method-test lines is uncommented:

```

$ python3 rangetest_test.py
-----Functions-----
Pat is 40 years old
Pat is 40 years old
birthday = 8/31/2024
-----Methods-----
sue=>120000, bob=>110000
TypeError: giveRaise argument "percent" not in 0.0..1.0

$ python3 -0 rangetest_test.py
...no error messages or default traces...

```

Implementation Details

This range-tester decorator's code relies on both introspection APIs and subtle constraints of argument passing. To be fully general, we could try to mimic Python's argument-matching logic in its entirety to see which names have been passed in which modes, but that's too much complexity for our tool and is prone to change over time. It would be better if we could somehow match the names of testable arguments given to the decorator against the names of actual arguments expected by the function to determine how the former map to the latter during a given call.

Function introspection

It turns out that the introspection API available on function objects and their associated code objects has exactly the tool we need. This API was briefly introduced in [Chapter 19](#), but we've actually put it to use here. The set of expected *argument names* is simply the first *N* variable names attached to a function's code object:

```
>>> def func(a, b, c, e=True, f=None):           # Args: three required, two defaults
    x = 1                                       # Plus two more local variables
    y = 2

>>> code = func.__code__                      # Code object of function object
>>> code.co_nlocals
7
>>> code.co_varnames                          # All local variable names
('a', 'b', 'c', 'e', 'f', 'x', 'y')
>>> code.co_varnames[:code.co_argcount]      # <== First N locals are expected args
('a', 'b', 'c', 'e', 'f')
```

And as usual, *starred-argument* names in the call proxy allow it to collect arbitrarily many arguments to be matched against the expected arguments so obtained from the function's introspection API:

```
>>> def catcher(*pargs, **kargs): print(f'{pargs}, {kargs}')

>>> catcher(1, 2, 3, 4, 5)
(1, 2, 3, 4, 5), {}
>>> catcher(1, 2, c=3, d=4, e=5)           # Arguments at calls
(1, 2), {'d': 4, 'e': 5, 'c': 3}
```

Run a `dir` call on function and code objects for more details.

Argument assumptions

Given the decorated function's set of expected argument names, the solution relies upon two constraints on argument passing *order* imposed by Python and covered in [Chapter 18](#):

- At the call, all positional arguments appear before all keyword arguments.
- In the `def`, all nondefault arguments appear before all default arguments.

That is, a nonkeyword argument cannot generally follow a keyword argument at a *call*, and a nondefault argument cannot follow a default argument at a *definition*. All *name=value* syntax must appear after any simple *name* in both places. As we've also learned, Python matches argument values passed by position to argument names in function headers from left to right, such that these values always match the *leftmost* names in headers. Keywords match by name instead, and a given argument can receive only one value.

To simplify our work, we can also make the assumption that a call is *valid* in general—that is, that all arguments either will receive values (by name or position) or will be omitted intentionally to pick up defaults. This assumption won't necessarily hold because the function has not yet actually been called when the wrapper logic tests validity—the call may still fail later when invoked by the wrapper layer due to incorrect argument passing. As long as that doesn't cause the wrapper to fail any worse, though, we can ignore the validity of the call. This helps because validating calls before they are actually made would require us to emulate Python's argument-matching algorithm in full.

Matching algorithm

Now, given these constraints and assumptions, we can allow for both keywords and omitted default arguments in the call with this algorithm. When a call is intercepted, we can make the following assumptions and deductions:

1. Let N be the number of passed positional arguments, obtained from the length of the `*pargs` tuple.
2. All N positional arguments in `*pargs` must match the first N expected arguments obtained from the function's code object. This is true per Python's call ordering rules, outlined earlier, since all positionals precede all keywords in a call.
3. To obtain the names of arguments actually passed by position, we can slice the list of all expected arguments up to the length N of the `*pargs` passed positionals tuple.
4. Any arguments after the first N expected arguments either were passed by keyword or were defaulted by omission at the call.
5. For each argument name to be validated by the decorator:
 - a. If the name is in `**kwargs`, it was passed by name—indexing `**kwargs` gives its passed value.
 - b. If the name is in the first N expected arguments, it was passed by position—its relative position in the expected list gives its relative position in `*pargs`.
 - c. Otherwise, we can assume it was omitted in the call and defaulted and need not be checked.

In other words, we can skip tests for arguments that were omitted in a call by assuming that the first N actually passed positional arguments in `*pargs` must match the first N argument names in the list of all expected arguments, and that any others must either have been passed by keyword and thus be in `**kwargs`, or have been defaulted. Under this scheme, the decorator will simply skip any argument to be checked that was omitted between the rightmost positional argument and the leftmost keyword argument, between keyword arguments, or after the rightmost positional in general. Trace through the decorator and its test script to see how this is realized in code.

Open Issues

Although our range-testing tool works as planned, three caveats remain—it doesn't detect invalid calls, doesn't handle some arbitrary-argument signatures, and doesn't fully support nesting. Improvements may require extension or altogether different approaches. Here's a quick rundown of the issues.

Invalid calls

First, as mentioned earlier, calls to the original function that are *not valid* still fail in our final decorator. The following, for example, both trigger `TypeError` exceptions for a missing positional argument `a`:

```
omitargs()
omitargs(d=8, c=7, b=6)
```

These only fail, though, where we try to *invoke* the original function, at the end of the wrapper. While we could try to imitate Python's argument matching to avoid this, there's not much reason to do so—since the call would fail at this point anyhow, we might as well let Python's own argument-matching logic detect the problem for us.

Arbitrary arguments

Second, although our final version handles positional arguments, keyword arguments, and omitted defaults, it still doesn't do anything explicit about **pargs* and ***kargs* starred-argument names that may be used in a decorated function `def` that accepts *arbitrarily many* arguments itself. This is probably moot for our purposes, though:

- If an extra *keyword* argument is passed, its name will show up in ***kargs* and can be tested normally if mentioned to the decorator.
- If an extra keyword argument is *not* passed, its name won't be in either ***kargs* or the sliced expected positionals list, and it will thus not be checked—it is treated as though it were defaulted, even though it is really an optional extra argument.
- If an extra *positional* argument is passed, there's no way to reference it in the decorator anyhow—its name won't be in either ***kargs* or the sliced expected arguments list, so it will simply be skipped. Because such arguments are not listed in the function's definition, there's no way to map a name given to the decorator back to an expected relative position.

In other words, as it is the code supports testing arbitrary keyword arguments by name, but not arbitrary positionals that are unnamed and hence have no set position in the function's argument signature. In terms of the function object's API, here's the effect of these tools in decorated functions:

```
>>> def func(*pargs, **kargs): pass
>>> code = func.__code__
>>> code.co_nlocals, code.co_varnames
(2, ('pargs', 'kargs'))
>>> code.co_argcount, code.co_varnames[:code.co_argcount]
(0, ())

>>> def func(a, b, *pargs, **kargs): pass
>>> code = func.__code__
>>> code.co_argcount, code.co_varnames[:code.co_argcount]
(2, ('a', 'b'))
```

Because starred-argument names show up as locals but *not* as expected arguments, they won't be a factor in our matching algorithm—names preceding them in function headers can be validated as usual, but not any extra positional arguments passed. In principle, we could extend the decorator's interface to support **pargs* in the decorated function, too, for the rare cases where this might be useful (e.g., a special argument name with a test to apply to all arguments in **pargs* beyond the length of the expected arguments list), but we'll pass on such an extension here.

Also, bear in mind that this pertains to values in starred *collectors* in `def` headers only; given that starred *unpackings* in *calls* are flattened before they ever reach our decorator, they are irrelevant to its code. To borrow a pathological example from [Chapter 18](#):

```
>>> def f(a, b, c, d, e, f, g, h, i): pass
>>> f.__code__.co_varnames[:f.__code__.co_argcount]
('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i')

>>> def f(*p, **k): print(p, k)
>>> f(*[1], 2, *[3], 4, f=6, *[5], **dict(g=7), h=8, **{'i': 9})
(1, 2, 3, 4, 5) {'f': 6, 'g': 7, 'h': 8, 'i': 9}
```

The call's stars here are resolved *before* the function is started. Because our decorator finds values passed to argument names by indexing keywords and mapping expected to actual positionals, it can remain blissfully ignorant of stars in the call and will work normally in this example (though, to be fair, “normally” may be an exaggeration here).

Decorator nesting

Finally, and perhaps most subtly, this code's approach does not fully support the use of *decorator nesting* to combine steps. Because it analyzes arguments using names in function definitions, and the names of the call proxy function returned by a nested decoration won't correspond to argument names in either the original function or decorator arguments, it does not fully support use in nested mode.

Technically, when nested, only the most deeply nested appearance's validations are run in full; all other nesting levels run tests on arguments passed by keyword only. Trace the code to see why; because the `onCall` proxy's call signature expects no named positional arguments, any to-be-validated arguments passed to it by position are treated as if they were omitted and hence defaulted and are thus skipped.

This may be inherent in this tool's approach—proxies change the argument name signatures at their levels, making it impossible to directly map names in decorator arguments to positions in passed argument sequences. When proxies are present, argument *names* ultimately apply to keywords only; by contrast, the first-cut solution's argument *positions* may support proxies better but do not fully support keywords.

In lieu of this nesting capability, we'll generalize this decorator to support *multiple kinds* of validations in a single decoration in an end-of-chapter quiz solution, which also gives examples of the nesting limitation in action. Since we've already neared the space allocation for this example, though, if you care about these or any other further improvements, you've officially crossed over into the realm of suggested exercises.

Decorator Arguments Versus Function Annotations

In closing, Python's annotation feature introduced in [Chapter 19](#) could also provide an alternative to the decorator arguments used by our example to specify range tests. As we learned earlier, annotations allow us to associate expressions with arguments and return values by coding them in the `def` header line itself; Python collects annotations in a dictionary and attaches it to the annotated function.

We could use this in our example to code range limits in the header line instead of in decorator arguments. We would still need a function decorator to wrap the function in order to intercept later calls, but we would essentially trade decorator argument syntax:

```
@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):
    print(a + b + c)
# func = rangetest(...)(func)
```

for annotation syntax like this:

```

@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)

```

That is, the range constraints would be moved into the function itself instead of being coded externally in a decorator line. [Example 39-27](#) illustrates the structure of the resulting decorators under both schemes in incomplete skeleton code for brevity. The decorator-arguments code pattern is that of our complete solution shown earlier; the annotations alternative requires one less level of nesting because it doesn't need to retain decorator arguments as state.

Example 39-27. *decoargs-vs-annotation.py*

Using decorator arguments

```

def rangetest(**argchecks):
    def onDecorator(func):
        def onCall(*pargs, **kargs):
            print(argchecks)
            for check in argchecks:
                pass
            return func(*pargs, **kargs)
        return onCall
    return onDecorator

@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):
    print(a + b + c)

func(1, 2, c=3)

```

Add validation code here

func = rangetest(...)(func)

Runs onCall, argchecks in scope

Using function annotations

```

def rangetest(func):
    def onCall(*pargs, **kargs):
        argchecks = func.__annotations__
        print(argchecks)
        for check in argchecks:
            pass
        return func(*pargs, **kargs)
    return onCall

@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)

func(1, 2, c=3)

```

Add validation code here

func = rangetest(func)

Runs onCall, annotations on func

When run, both schemes have access to the same validation test information but in different forms—the decorator argument version's information is retained in an argument in an enclosing scope, and the annotation version's information is retained in an attribute of the function itself:

```

$ python3 decoargs-vs-annotation.py
{'a': (1, 5), 'c': (0.0, 1.0)}
6
{'a': (1, 5), 'c': (0.0, 1.0)}
6

```

Fleshing out the rest of the annotation-based version is left as a suggested exercise; its code would be almost identical to that of our earlier solution because range-test information is simply on the function

instead of in an enclosing scope. Really, all this buys us is a different user interface for our tool—it will still need to match argument names against expected argument names to obtain relative positions as before.

In fact, using annotation instead of decorator arguments in this example actually *limits its utility*. By moving the validation specifications into the `def` header, we essentially commit the function to a *single role*—since annotation directly allows us to code only one expression per argument, it can have only one purpose. For instance, we cannot use range-test annotations for any other role (including the optional and unused type hinting of [Chapter 6](#)).

By contrast, because decorator arguments are coded outside the function itself, they are both easier to remove and *more general*—the code of the function itself does not imply a single decoration purpose. Crucially, by *nesting* decorators with arguments, we can often apply multiple augmentation steps to the same function; annotation directly supports only one. With decorator arguments, the function itself also retains a simpler, normal appearance.

Still, if you have a single purpose in mind, the choice between annotation and decorator arguments is largely stylistic and subjective. As is so often true in life, one person’s decoration or annotation may well be another’s syntactic clutter.

Chapter Summary

In this chapter, we explored decorators—both the function and class varieties. As we learned, decorators are a way to insert code to be run automatically when a function or class is defined. When a decorator is used, Python rebinds a function or class name to the callable object that the decorator returns. This hook allows us to manage functions and classes themselves or later calls to them. By adding a layer of wrapper logic to catch later calls, we can augment both function calls and instance interfaces. Decorators provide an explicit and uniform way to achieve such goals.

As we also learned, class decorators can be used to manage classes themselves rather than just their instances. Because this functionality overlaps with *metaclasses*—the topic of the next and final technical chapter—you’ll have to read on for the conclusion to this story and that of this book at large.

First, though, let’s work through the following quiz. Because this chapter was mostly focused on its examples, the quiz will ask you to modify some of its examples’ code in order to review their concepts. Both the examples’ code and the quiz’s solutions are located in the book’s *examples package* (see the [Preface](#) for access pointers). Look for the solutions’ code there in this chapter’s `_QuizAnswers` subfolder. If you’re pressed for time, you’re welcome to jump right into studying the solutions; programming is often as much about reading code as writing it.

Test Your Knowledge: Quiz

1. *Method decorators*: As mentioned in one of this chapter’s notes, the `timerdeco2.py` module’s call-timer decorator that we wrote in [Example 39-10](#) of “Adding Decorator Arguments” on [page 1055](#) can be applied only to simple *functions* because it uses a nested class with a `__call__` operator-overloading method to catch calls. This structure does not work for a class’s *methods* because the *decorator* instance is passed to `self`, not the subject-class instance. Rewrite this decorator so that it can be applied to *both* simple functions and methods in classes, and test it on both functions and methods. (Hint: see “Class Pitfall: Decorating Methods” on [page 1048](#) for pointers.) Note that you will probably need to use function-object *attributes* to keep track of total time, since you won’t have a nested class for state retention and can’t access nonlocals from outside the decorator code.

2. *Class decorators*: The `Public/Private` class decorators we wrote in module `access2.py` of [Example 39-18](#) in this chapter's first case study example will add *performance costs* to every attribute fetch in a decorated class. Although we could simply delete the `@` decoration line to gain speed, we could also augment the decorator itself to check the `__debug__` switch and perform no wrapping at all when the `-O` Python flag is passed on the command line—just as we did for the argument range-test decorators. That way, we can speed our program without changing its source via command-line arguments (`python -O main.py`). While we're at it, we could also use one of the mix-in superclass techniques we studied to catch a few *built-in operations* too. Code and test these two extensions.
3. *Generalized argument validations*: The function and method decorator we wrote in `rangetest.py` of [Example 39-25](#) checks that passed arguments are in a valid range, but the same code pattern could apply to similar goals such as argument type testing and possibly more. Generalize the range tester so that its single code base can be used for *multiple kinds* of argument validations. Passed-in validation functions may be the simplest solution given the coding structure here, though subclasses that provide expected methods can often provide similar generalization routes as well. This is substantially challenging, so be sure to see the solution for tips.

Test Your Knowledge: Answers

As noted, coding solutions for this quiz are in this chapter's `_QuizAnswers` subfolder of the book examples package. Each question has its own subfolder there for its files, with a `_Notes.txt` plain-text file giving background info. This edition opted to move these solutions online instead of listing them here because it saves about 10 pages and because this internet thing just might take off after all.

Metaclasses and Inheritance

In [Chapter 39](#), we explored decorators and studied examples of their use. In this final technical chapter of the book, we're going to continue our tool-builders focus with an in-depth review of another advanced topic: *metaclasses*, a protocol for managing class objects instead of their instances, introduced briefly in [Chapter 32](#).

On a base level, metaclasses extend the code-insertion model of decorators. As we learned in the prior chapter, decorators allow us to augment functions and classes by intercepting their creation. Metaclasses similarly allow us to intercept and augment *class creation*—they provide a hook for inserting extra logic to be run at the conclusion of a `class` statement, albeit in different ways than decorators.

Metaclasses can also provide behavior for classes with *methods* located in a separate inheritance tree skipped for normal, nonclass instances. While this allows metaclasses to process their instance classes after creation, it also compounds class semantics and convolutes *inheritance*—whose full definition can finally be fleshed out here.

Like all the subjects covered in this part of the book, this is an *advanced topic* that can be studied on an as-needed basis. Metaclasses are not generally in scope for most application programmers but may be of interest to others seeking to write flexible tools. Whatever category you fall into, though, metaclasses can teach you more about Python's classes and are a prerequisite to both code that employs them and the complete inheritance story in Python.

As the last technical chapter of this book, this also begins to wrap up some threads concerning Python itself that we have met often along the way and will finalize in the conclusion that follows. Where you go after this book is up to you, but in an open source project, it's important to keep the big picture in mind while hacking the small details.

To Metaclass or Not to Metaclass

Despite the advanced status awarded to metaclasses in the preceding opener, they have a variety of potential roles. For example, they can be used to enhance classes with features like tracing, object persistence, exception logging, and more. They can also be used to construct portions of a class at runtime based on configuration files, apply function decorators to every method of a class generically, verify conformance to expected interfaces, and so on.

In their more grandiose incarnations, metaclasses can even be used to implement alternative coding patterns such as aspect-oriented programming, object/relational mappers (ORMs) for databases, and more. Although there are often alternative ways to achieve such results—as you’ll see, the roles of class decorators and metaclasses often intersect—metaclasses provide a formal model tailored to those tasks. We don’t have space to explore all such applications first-hand in this chapter, of course, but you can find additional use cases on the web after studying the basics here.

Probably the reason for studying metaclasses most relevant to this book is that this topic can help demystify Python’s class mechanics in general. For instance, you’ll find that they are an intrinsic part of the language’s inheritance model formalized in full here. Although you may or may not code or reuse them in your work, a cursory understanding of metaclasses can impart a deeper understanding of Python at large.

The Downside of “Helper” Functions

Before we get to metaclass code, let’s get a better handle on its rationale. Like the decorators of the prior chapter, metaclasses are optional in principle. We can usually achieve the same effect by passing class objects through functions—known interchangeably as *helper* or *manager* functions—much as we can achieve the goals of decorators by passing functions and classes through manager code. Just like decorators, though, metaclasses:

- Provide a more uniform and explicit structure
- Help ensure that application programmers won’t forget to augment their classes according to an API’s requirements
- Avoid code redundancy and its associated maintenance costs by factoring class customization logic into a single location

To illustrate, suppose we want to automatically insert a method into a set of classes. Of course, we could do this with simple *inheritance* if the subject method is known when we code the classes. In that case, we can simply code the method in a superclass and have all the classes in question inherit from it:

```
class Extras:
    def extra(self, args):          # Normal inheritance: too static
    ...

class Client1(Extras): ...         # Clients inherit extra methods
class Client2(Extras): ...

X = Client1()                     # Make an instance
X.extra()                         # Run the extra methods
```

Sometimes, though, it's impossible to predict such augmentation when classes are coded. Consider the case where classes are augmented in response to choices made in a user interface at runtime or loaded from an editable configuration file. Although we could code every class in our imaginary set to *manually* check these, too, it's a lot to ask of clients (the required function here is abstract—it's something to be filled in):

```
def extra(self, arg): ...

class Client1: ...                # Client augments: too distributed
if required():
    Client1.extra = extra

class Client2: ...
if required():
    Client2.extra = extra        # Add the extra method - maybe

X = Client1()
X.extra()
```

We can add methods to a class after the `class` statement like this because, as we've learned, a class-level method is just a plain function that is associated with a class and has a first argument to receive a `self` instance when called through one. Although this works, it might become untenable for larger method sets and puts all the burden of augmentation on each client class (and assumes they'll remember to do this at all).

It would be better from a maintenance perspective to isolate the decision logic in a single place. We might encapsulate some of this extra work by routing classes through a *helper function*—a function that would extend the class as required and handle all the work of runtime testing and configuration:

```
def extra(self, arg): ...

def extras(Class):                # Helper function: too manual
    if required():
        Class.extra = extra

class Client1: ...
extras(Client1)

class Client2: ...
extras(Client2)

X = Client1()
X.extra()
```

This code runs the class through a helper function immediately after it is created. Although functions like this one can achieve our goal here, they still put a burden on class coders, who must understand the requirements and adhere to them in their code. It would be even better if there was a simple way to enforce the augmentation in the subject classes, so that they don't need to deal with the augmentation so explicitly and would be less likely to forget to use it altogether. In other words, we'd like to be able to insert some code to run *automatically* at the end of a `class` statement to augment the class.

This is exactly what *metaclasses* do—by declaring a metaclass, we tell Python to route the creation of the class object to another class we provide:

```

def extra(self, arg): ...

class Extras(type):
    def __init__(Class, classname, superclasses, attributedict):
        if required():
            Class.extra = extra

class Client1(metaclass=Extras): ...      # Metaclass declaration only
class Client2(metaclass=Extras): ...      # Client class is instance of meta

X = Client1()                            # X is instance of client class
X.extra()

```

Because Python invokes the metaclass automatically at the end of the `class` statement when the new class is *created*, it can augment, register, wrap, or otherwise manage the class as needed. Moreover, the only requirement for the client classes is that they *declare* their metaclass; every class that does so will automatically acquire whatever augmentation the metaclass provides, both now and in the future if the metaclass changes.

Metaclasses can also augment their class instances with inherited *methods*, which are akin to normal class methods but not inherited by the instances of their class instances—an inheritance extension and tongue twister whose true nature requires more info ahead (and quite possibly, sedation). Through both changes and methods, though, metaclasses can customize class behavior broadly.

Of course, this is the standard rationale, which you'll need to judge for yourself—in truth, clients might forget to list a metaclass just as easily as they could forget to call a helper function! Still, the explicit nature of metaclasses may make this less likely. Although it may be difficult to glean from this small and hypothetical example, metaclasses generally handle such tasks better than more manual approaches.

Metaclasses Versus Class Decorators: Round 1

Having said that, it's also important to note that the *class decorators* described in the preceding chapter sometimes overlap with metaclasses—in terms of both utility and benefit. Like metaclasses, class decorators can be used to manage both classes and their later instances, and their syntax makes their usage similarly explicit and arguably more obvious than helper-function calls.

For example, suppose we recoded the last section's helper function to *return* the augmented class instead of simply modifying it in place. This would allow a greater degree of flexibility because the manager would be free to return any type of object that implements the class's expected interface:

```

def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class      # Return the augmented class

class Client1: ...
Client1 = extras(Client1)      # Rebind to augmented class

class Client2: ...
Client2 = extras(Client2)

X = Client1()
X.extra()

```

If you think this is starting to look reminiscent of class decorators, you're right. In the prior chapter, we emphasized class decorators' role in augmenting *instance* creation calls. Because they work by automatically rebinding a class name to the result of a function, though, there's no reason that we can't use them to augment the class by changing it before any instances are ever created. That is, class decorators can apply extra logic to *classes*, not just *instances*, at class creation time:

```
def extra(self, arg): ...

def extras(Class):           # From helper to decorator
    if required():
        Class.extra = extra
    return Class

@extras
class Client1: ...           # Client1 = extras(Client1)

@extras
class Client2: ...           # Rebinds class independent of instances

X = Client1()                # Makes instance of augmented class
X.extra()                    # X is instance of original Client1
```

Decorators essentially automate the prior example's manual name rebinding here. Just as for metaclasses, because this decorator returns the original class, instances are made from that class, not from a wrapper object. In fact, instance creation is not intercepted at all in this example.

In this specific case—adding methods to a class when it's created—the choice between metaclasses and decorators is arbitrary. Decorators can be used to manage both instances and classes and intersect most strongly with metaclasses in the second of these roles, but this discrimination is not absolute. In fact, the roles of each are suggested in part by their mechanics.

As we'll detail ahead, decorators technically correspond to metaclass *calls* used to make and initialize new classes. Metaclasses, though, have additional customization hooks beyond class creation. Their *methods* inherited by classes, for example, have no direct counterpart in class decorators sans extra code. This can make metaclasses more complex but also better suited for augmenting classes in some contexts.

Conversely, because metaclasses are designed to manage classes, applying them to managing *instances* alone is less optimal. Because they are also responsible for making the class itself, metaclasses incur this as an *extra* step in instance-management roles and are perhaps less appropriate than decorators.

We'll explore some of these differences in working code later in this chapter. To better understand how metaclasses do their work, though, we first need to get a clearer picture of their underlying model.

The Metaclass Model

To understand metaclasses, you first need to understand a bit more about both Python's object model and what happens at the end of a `class` statement. As you'll learn here, the two are intimately related.

Classes Are Instances of type

The first of these prerequisites was covered previously by “[The Python Object Model](#)” on page 821, which in turn assumes knowledge of the *MRO* (method resolution order) covered in [Chapter 31](#). You should review that content now if needed, especially if you've jumped into this chapter at random. We're

not going to repeat its coverage in full here, but some of its conclusions are crucial to understanding metaclasses. Namely:

- Instances are created from classes.
- Classes are instances of a metaclass.
- The `type` built-in is the topmost metaclass.
- Metaclasses customize `type` with normal `class` statements.

In short, classes are types, types are classes, and metaclasses customize types. Per [Chapter 32](#), the relationship between metaclasses and classes is subtly different from that between classes and their *nonclass* instances. The latter do not generate more instances, and while attribute inheritance uses the same core mechanisms and MRO everywhere, classes search a metaclass tree that nonclass instances do not.

We'll study the nuts and bolts of inheritance ahead, but it's easy to see this model's fundamentals in code. Built-in objects like lists are actually nonclass instances made from a class, which itself is made from the built-in `type`:

```
>>> type([]), type(type([]))      # List instance is created from list class
(<class 'list'>, <class 'type'>)   # List class is created from type class

>>> type(list), type(type)        # Same, but with type names
(<class 'type'>, <class 'type'>)   # Type of type is type: top of hierarchy
```

Apart from the literal syntax of built-ins, this works the same way for user-defined classes, which are really just user-defined types—their nonclass instances are made from the class, which itself is made from the built-in `type`:

```
>>> class Hack: pass              # User-defined classes work the same
>>> I = Hack()                   # Made from class, which is made from type

>>> type(I), type(type(I))
(<class '__main__.Hack'>, <class 'type'>)
```

Although their behavior varies in ways we explored in [Chapter 32](#), both classes and nonclass instances are “instances” in some sense. In fact, the `__class__` attribute in both tells us what they were made from:

```
>>> [].__class__, list.__class__  # Instances and classes are both "instances"
(<class 'list'>, <class 'type'>)  # type(X) is normally same as X.__class__

>>> I.__class__, Hack.__class__
(<class '__main__.Hack'>, <class 'type'>)
```

Because classes are created from the root `type` class by default, most programmers don't need to think about this model. However, it's key to understanding the way that metaclasses work—as the next section explains.

Metaclasses Are Subclasses of `type`

Why would we care that classes are instances of a `type` class? It turns out that this is the hook that allows us to code metaclasses. Specifically, we can create classes from *subclasses* of `type` that customize it with normal object-oriented techniques and class syntax. And these `type` subclasses are known as *metaclasses*.

In other words, to control the way classes are created and augment their behavior, all we need to do is specify that a user-defined class be created from a user-defined metaclass instead of the normal and default type class.

Before we see how, it's important to bear in mind that this *type instance* relationship is not quite the same as normal *inheritance*. User-defined classes may also have *superclasses* from which they and their instances inherit attributes as usual. As we've seen, inheritance superclasses are listed in parentheses in the class statement, show up in a class's `__bases__` tuple, and are searched for attributes fetched from nonclass instances.

However, the type from which a class is created and of which it is an instance is a different relationship. Inheritance searches instance and class namespace dictionaries, but classes may also acquire behavior from their type that is not exposed to the normal inheritance search. In fact, metaclasses define a separate, *secondary* inheritance tree available only to classes and used as a fallback when the normal superclass search fails.

To lay the groundwork for understanding this distinction, the next section describes the procedure and syntax Python uses to implement this *instance-of* relationship.

Class Statements Call a type

Subclassing the type class to customize it is really only half of the metaclass backstory. We still need to somehow route a class's creation to the metaclass instead of the default type. To comprehend the way that this is arranged, we also need to know how class statements do their business.

We've already learned that when Python reaches a class statement, it runs its nested block of code to create the class's attributes—all the names assigned at the top level of the nested code block generate attributes in the resulting class object. These names are usually method functions created by nested defs, but they can also be arbitrary attributes assigned to create class data shared by all instances.

Technically speaking, Python follows a standard protocol to make this happen: at the *end* of a class statement, and after running all its nested code in a namespace dictionary corresponding to the class's local scope, Python calls the type object to create the new class object like this:

```
class = type(classname, superclasses, attributedict)
```

The type object in turn defines a `__call__` operator-overloading method that runs two other methods when the type object is called:

```
type.__new__(typeclass, classname, superclasses, attributedict)
type.__init__(class, classname, superclasses, attributedict)
```

The `__new__` method creates and returns the new class object, after which the `__init__` method initializes the newly created object. As you'll see in a moment, these are the hooks that metaclass subclasses of type generally use to perform class customizations at creation time.

For example, given a class definition like the following for Hack:

```
class Super: ...                # Inherited names here

class Hack(Super):              # Inherits from Super
    data = 1                    # Class data attribute
    def meth(self, arg):        # Class method attribute
        return self.data + arg
```

Python will internally run the nested code block to create two attributes of the class (`data` and `meth`), and then call the type object to generate the class object at the end of the `class` statement's processing (extra names like `__module__` are added automatically from the code's context):

```
Hack = type('Hack', (Super,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

In fact, you can call `type` this way yourself to create a class *dynamically*—albeit here with a fabricated method function and empty superclasses tuple (Python adds the object superclass automatically to topmost classes as we learned in [Chapter 32](#), and the enclosing module's name is again implied):

```
>>> c = type('Hack', (), {'data': 1, 'meth': (lambda x, y: x.data + y)})
>>> i = c()
>>> c, i
(<class '__main__.Hack'>, <__main__.Hack object at 0x108077c20>)
>>> i.data, i.meth(2)
(1, 3)
```

The class produced by a direct `type` call is exactly like that you'd get from running a `class` statement (again, if you've forgotten what some of the following are about, flip, click, or tap back to [Chapter 32](#) for a refresher):

```
>>> c.__bases__
(<class 'object'>,)
>>> i.__class__.__mro__
(<class '__main__.Hack'>, <class 'object'>)

>>> [a for a in dir(i) if not a.startswith('__')]
['data', 'meth']

>>> [(a, v) for (a, v) in c.__dict__.items() if not a.startswith('__')]
[('data', 1), ('meth', <function <lambda> at 0x1082179c0>)]
```

Because this `type` call is made automatically at the end of the `class` statement, though, it's an ideal hook for augmenting or otherwise processing a class. The trick lies in replacing the default `type` with a custom subclass that will intercept this call. The next section shows how.

Class Statements Can Choose a type

As we've just seen, classes are created by the `type` class by default. To tell Python to create a class with a custom *metaclass* instead, you simply need to declare a metaclass to intercept the normal instance creation call for a user-defined class. To do so, list the desired metaclass as a keyword argument in the `class` header:

```
class Hack(metaclass=Meta):                                # Use Meta instead of type default
```

If no such declaration is present, the metaclass to be called defaults to the `type` built-in, per the prior section. When used, though, this declaration overrides the `type` default and routes the class creation call at the close of the `class` statement to *Meta* instead:

```
class = Meta(classname, superclasses, attributedict)
```

Importantly again, the `metaclass` keyword specifies an *instance-of* relationship, which implies inheritance only through the secondary metaclass tree we'll formalize ahead. Normal *inheritance* superclasses can be listed in the header as well and take precedence by residing in the primary class tree. In the following, for example, the new class `Hack` inherits from superclass `Super` normally but is also an instance of—and is created by—metaclass *Meta*:

```
class Hack(Super, metaclass=Meta):                          # Normal supers OK: listed first
```

In this form, superclasses must be listed before the metaclass; in effect, the ordering rules used for keyword arguments in function calls apply here too. This order also has implications for inheritance, which we'll formalize soon, but first, we need to learn how to code the metaclasses that tap into calls triggered by this special `class` syntax.

Metaclass Method Protocol

When a specific metaclass is declared per the prior sections' syntax, the call to create the `class` object run at the end of the `class` statement is modified to invoke the *metaclass* instead of the `type` default, as we just saw:

```
class = Meta(classname, superclasses, attributedict)
```

Assuming the metaclass is a subclass of `type`, though, the `type` class's inherited `__call__` method delegates creation and initialization of the new `class` object to the metaclass if the metaclass defines custom versions of the methods that handle these steps. In other words, the *Meta* call may wind up triggering these method calls in turn:

```
class = Meta.__new__(Meta, classname, superclasses, attributedict)
Meta.__init__(class, classname, superclasses, attributedict)
```

To demonstrate, here's the preceding class example again, augmented with a metaclass specification:

```
class Hack(Super, metaclass=Meta):    # Inherits from Super, instance of Meta
    data = 1                          # Class data attribute
    def meth(self, arg):              # Class method attribute
        return self.data + arg
```

At the end of this `class` statement, Python internally runs the following to create the `class` object—again, a call you could make manually, too, but automatically run by Python's class machinery:

```
Hack = Meta('Hack', (Super,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

If the metaclass defines its own versions of `__new__` or `__init__`, they will be invoked during this call by the inherited `type` class's `__call__` method. The net effect is to automatically run methods the metaclass provides as part of the class-construction process. The next section shows how we might go about coding this final piece of the metaclass puzzle.

Coding Metaclasses

So far, we've seen how Python routes class creation calls to a metaclass if one is specified and provided. How, though, do we actually *code* a metaclass that customizes `type`?

It turns out that you already know most of the story—metaclasses are coded with normal Python `class` statements and semantics. By definition, they are simply classes that inherit from `type` (normally, at least). Their only substantial distinctions are that Python calls them *automatically* at the end of a `class` statement and that they must generally adhere to the *interface* expected by the `type` superclass if they subclass it.

A Basic Metaclass

Perhaps the simplest metaclass you can code is simply a subclass of `type` with a `__new__` method that creates the class object by running the default method in `type`. A metaclass `__new__` like this is run by the `__call__` method inherited from `type` by virtue of normal inheritance overrides; this method

typically performs whatever augmentation is required and calls the type superclass's `__new__` method to create and return the new class object:

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):
        # Run by inherited type.__call__
        return type.__new__(meta, classname, supers, classdict)
```

This metaclass doesn't really do anything (we might as well let the default type create the class), but it demonstrates the way a metaclass taps into the metaclass hook to customize—because the metaclass is called at the end of a class statement, and because the type object's `__call__` dispatches to the `__new__` and `__init__` methods, code we provide in these methods can manage all the classes created from the metaclass.

To demo, [Example 40-1](#) is our inane metaclass again, but in more tangible form, with prints added to the metaclass and the file at large to trace the process.

Example 40-1. *metaclass1.py*

```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
        print('In MetaOne.new:', meta, classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)

class Super:
    pass

print('Making class')
class Hack(Super, metaclass=MetaOne):    # Inherits from Super, instance of MetaOne
    data = 1                             # Class data attribute
    def meth(self, arg):                 # Class method attribute
        return self.data + arg

print('Making instance')
X = Hack()
print('Attrs:', X.data, X.meth(2))
```

Here, class `Hack` inherits from `Super` and is an instance of `MetaOne`, but `X` is an instance of and inherits from `Hack`. When run, notice how the metaclass is invoked at the *end* of the class statement and before we ever make an instance of the new class—*metaclasses process classes*, and classes process *nonclass* instances:

```
$ python3 metaclass1.py
Making class
In MetaOne.new:
...<class '__main__.MetaOne'>
...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function Hack.meth at 0x...>}
Making instance
Attrs: 1 3
```

Presentation note: this chapter's examples often truncate hex addresses and omit some irrelevant built-in `__X__` names in namespace dictionaries, both for brevity and because built-in attributes tend to change over time. Run these examples on your own for full, if transient, fidelity.

Customizing Construction and Initialization

Metaclasses can also tap into the `__init__` protocol invoked by the type object's `__call__`. In general, `__new__` creates and returns the class object, and `__init__` initializes the already created class passed in as an argument. These methods work in *non-type* classes, too, but `__new__` is rare in such classes. Metaclasses can use either or both hooks to manage classes at creation time, as [Example 40-2](#) illustrates.

Example 40-2. *metaclass2.py*

```
class MetaTwo(type):
    def __new__(meta, classname, supers, classdict):
        print()
        print('In MetaTwo.new:', meta, classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)

    def __init__(Class, classname, supers, classdict):
        print()
        print('In MetaTwo.init:', Class, classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Super:
    pass

print('Making class')
class Hack(Super, metaclass=MetaTwo):    # Inherits from Super, instance of MetaTwo
    data = 1                             # Class data attribute
    def meth(self, arg):                 # Class method attribute
        return self.data + arg

print('\nMaking instance')
X = Hack()
print('Attrs:', X.data, X.meth(2))
```

In this case, the class *initialization* method is run after the class *construction* method, but both methods run at the end of the class statement and before any nonclass instances are made. Conversely, an `__init__` in Hack would run later at *nonclass-instance* creation time and would not be affected or run by the metaclass's `__init__`:

```
$ python3 metaclass2.py
Making class
In MetaTwo.new:
...<class '__main__.MetaTwo'>
...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function Hack.meth at 0x...>}

In MetaTwo.init:
...<class '__main__.Hack'>
...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function Hack.meth at 0x...>}
...init class object: ['__module__', 'data', 'meth', '__doc__']

Making instance
Attrs: 1 3
```

Other Metaclass Coding Techniques

Although redefining the type superclass's `__new__` and `__init__` methods is the most common way to insert logic into the class object creation process with the metaclass hook, other schemes are possible. They may not dovetail as neatly into the notion of metaclass methods we'll study ahead, but they do support creation-time tasks.

Using simple factory functions

For example, metaclasses need not really be classes at all. As we've learned, the `class` statement issues a simple call to create a class at the conclusion of its processing. Because of this, *any callable object* can, in principle, be used as a metaclass, provided it accepts the arguments passed and returns an object compatible with the intended class. In fact, a simple object factory function may serve just as well as a type subclass, as [Example 40-3](#) demonstrates.

Example 40-3. *metaclass3.py*

```
# A simple function can serve as a metaclass too

def MetaFunc(classname, supers, classdict):
    print('In MetaFunc:', classname, supers, classdict, sep='\n...')
    return type(classname, supers, classdict)

class Super:
    pass

print('Making class')
class Hack(Super, metaclass=MetaFunc):           # Run simple function at end
    data = 1                                     # Function returns class
    def meth(self, arg):
        return self.data + arg

print('Making instance')
X = Hack()
print('Attrs:', X.data, X.meth(2))
```

When run, the function is called at the end of the declaring `class` statement, and it returns the expected new class object. The function is simply catching the call that the type object's `__call__` normally intercepts by default:

```
$ python3 metaclass3.py
Making class
In MetaFunc:
...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function Hack.meth at 0x...>}
Making instance
Attrs: 1 3
```

Technically speaking, such a plain function used as a metaclass can return *anything*: whatever it returns is assigned to the new class's name, whether it's a type instance or not. Other kinds of results may not support later instance creation, but this blurs the distinction between metaclasses and class decorators—both rebound names in the end.

Overloading class creation calls with normal classes

Because normal (a.k.a. *nonclass*) instances can respond to call operations with operator overloading, they can serve in some metaclass roles, too, much like the preceding function. The output of [Example 40-4](#) is similar to the prior class-based versions, but it's based on a simple class—one that doesn't inherit from `type` at all and provides a `__call__` for its instances that catches the metaclass call using normal operator overloading.

All classes are created from a metaclass, even if it's the default `type` metaclass, but here we're using a *nonclass instance* of a class for the “metaclass.” Note that `__new__` and `__init__` must use different names here, or else they will run when the `MetaObj` instance is *created*, not when that instance is later called in the role of metaclass after Hack's class statement. The `__call__` here mimics part of what `type`'s method does.

Example 40-4. *metaclass4.py*

A normal class instance can serve as a metaclass too

```
class MetaObj:
    def __call__(self, classname, supers, classdict):
        print('In MetaObj.call:', classname, supers, classdict, sep='\n...')
        Class = self.__New__(classname, supers, classdict)
        self.__Init__(Class, classname, supers, classdict)
        return Class

    def __New__(self, classname, supers, classdict):
        print('In MetaObj.new: ', classname, supers, classdict, sep='\n...')
        return type(classname, supers, classdict)

    def __Init__(self, Class, classname, supers, classdict):
        print('In MetaObj.init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Super:
    pass

print('Making class')
class Hack(Super, metaclass=MetaObj()):          # MetaObj() is normal class instance
    data = 1                                    # Called at end of statement
    def meth(self, arg):
        return self.data + arg

print('Making instance')
X = Hack()
print('Attrs:', X.data, X.meth(2))
```

When run, the three methods are dispatched via the normal instance's `__call__` inherited from its normal class, but without any dependence on type dispatch mechanics or semantics. This routing is largely about class alone:

```
$ python3 metaclass4.py
Making class
In MetaObj.call:
...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function Hack.meth at 0x...>}
In MetaObj.new:
```

```

...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function Hack.meth at 0x...>}
In MetaObj.init:
...Hack
...(<class '__main__.Super'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function Hack.meth at 0x...>}
...init class object: ['__module__', 'data', 'meth', '__doc__']
Making instance
Attrs: 1 3

```

In fact, we can use normal superclass inheritance to acquire the call interceptor in this coding model—the superclass in [Example 40-5](#) serves essentially the same role as `type`, at least in terms of metaclass dispatch. Such code may be atypical, but it demos the underlying metaclass dispatch model.

Example 40-5. *metaclass5.py*

Instances inherit from classes and their supers normally

```

class SuperMetaObj:
    def __call__(self, classname, supers, classdict):
        print('In SuperMetaObj.call:', classname, supers, classdict, sep='\n...')
        Class = self.__New__(classname, supers, classdict)
        self.__Init__(Class, classname, supers, classdict)
        return Class

class SubMetaObj(SuperMetaObj):
    def __New__(self, classname, supers, classdict):
        print('In SubMetaObj.new: ', classname, supers, classdict, sep='\n...')
        return type(classname, supers, classdict)

    def __Init__(self, Class, classname, supers, classdict):
        print('In SubMetaObj.init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Super:
    pass

print('Making class')
class Hack(Super, metaclass=SubMetaObj()): # Invoke Sub instance via Super.__call__
    ...rest of file same as Example 40-4...

```

This example’s output is largely the same as that of its predecessor but reflects normal inheritance at work:

```

$ python3 metaclass5.py
Making class
In SuperMetaObj.call:
...as before...
In SubMetaObj.new:
...as before...
In SubMetaObj.init:
...as before...
Making instance
Attrs: 1 3

```

Although such alternative forms work, most metaclasses achieve their creation-time goals by redefining the type superclass’s `__new__` and `__init__`; in practice, this may be simpler than other schemes.

Regardless of its coding, though, this metaclass role broadly intersects with class decorators—as the next section will demonstrate.

Managing Classes with Metaclasses and Decorators

Now that we understand the hook metaclasses use to insert code to be run at class construction time, let's put it to better use. In general, such code can be used to augment classes arbitrarily, and in many of the same ways as the *class decorators* of [Chapter 39](#). This doesn't make these two tools identical—metaclasses also support the notion of inherited methods coming up later—but they are functionally redundant in some roles.

Adding methods to classes

To demo both this equivalence and more realistic usage, [Example 40-6](#) uses metaclasses to augment the set of methods available in classes by *inserting* new methods at class-creation time as sketched in the abstract earlier—not the sort of thing most programmers do on a day-to-day basis, but potentially useful in tools and libraries nonetheless.

Example 40-6. extend_meta.py

"Extend a class with a metaclass"

```
def triple(obj):
    return obj.value * 3                                # Functions to insert
                                                         # Methods if in a class
                                                         # Where "obj" is "self"

def concat(obj):
    return obj.value + 'Code!'

class Extender(type):
    def __new__(meta, classname, supers, classdict):    # On client-class creation
                                                         # Add funcs as attributes
        classdict['triple'] = triple
        classdict['concat'] = concat
        return type.__new__(meta, classname, supers, classdict)

class Client1(metaclass=Extender):
    def __init__(self, value):                          # Created from Extender
                                                         # Own + inserted methods
        self.value = value
    def double(self):
        return self.value * 2

class Client2(metaclass=Extender):
                                                         # Created from Extender
                                                         # Inherited class data
    value = 'grok'

X = Client1('hack')
print(X.double(), X.triple(), X.concat(), sep='\n')

Y = Client2()
print(Y.triple(), Y.concat(), sep='\n')
```

Recall again that class methods are simply *functions* that normally receive an instance through which they are called. The metaclass in this code leverages this to insert two functions into each of its client classes when those classes are made. The net effect provides methods and behavior for clients that do not exist in their class statements:

```
$ python3 extend_meta.py
hackhack
hackhackhack
hackCode!
grokgrokgrok
grokCode!
```

Of course, the methods inserted here might be coded in a *superclass* and inherited by clients as usual, but the metaclass here is free to select inserted methods based on conditions tested whenever clients are built. As covered ahead, we can also *almost* do the same with metaclass *methods*, but these methods are inherited only by classes and wouldn't work in this example; the methods inserted here are instead available to later class instances like X and Y.

And while this may seem novel, it's easy to accomplish the same result with class decorators. As we've learned, there are indeed some striking similarities between these tools:

- *Class decorators* work by rebinding class names to the result of a callable at the end of a `class` statement after the new class has been created.
- *Metaclasses* work by routing class-object creation through a callable at the end of a `class` statement in order to create the new class.

The sum makes these two tools functionally equivalent, at least in terms of class-creation dispatch. **Example 40-7** illustrates this equivalence by recoding the same augmentation with a class decorator instead of a metaclass.

Example 40-7. `extend_deco.py`

"Extend a class with a decorator"

```
def triple(obj):
    return obj.value * 3

def concat(obj):
    return obj.value + 'Code!'

def extender(aClass):
    aClass.triple = triple          # Manages class, not instance
    aClass.concat = concat         # Same as metaclass __call__
    return aClass

@extender
class Client1:                    # Client1 = Extender(Client1)
    def __init__(self, value):    # Rebound at end of class stmt
        self.value = value
    def double(self):
        return self.value * 2

@extender
class Client2:
    value = 'grok'

X = Client1('hack')
print(X.double(), X.triple(), X.concat(), sep='\n')

Y = Client2()
print(Y.triple(), Y.concat(), sep='\n')
```

When run, this class-decorator version's output is identical to that of the metaclass variant in [Example 40-6](#). It works the same because both decorator and metaclass are called at the end of a class statement and return an object to which the class's name is assigned. Decorators may be closest to the metaclass `__call__` because they are called to return an object, but like metaclass `__init__`, the class has already been built by the time a decorator runs. Metaclass `__call__` runs `__new__` and `__init__`, and metaclasses can augment in any of these three methods.

Automatically decorating class methods

Perhaps more interesting, metaclasses and decorators can both augment individual methods of classes. To demo, we'll use the utility module in [Example 40-8](#), which resurrects the tracer and timer function decorators we coded in the prior chapter. This module is entirely review, so we'll defer to [Chapter 39](#) for more details (and promise that this is the last mileage we'll get from this code).

Example 40-8. decorators.py

```
import time

def tracer(func):
    calls = 0
    def onCall(*args, **kwargs):
        nonlocal calls
        calls += 1
        print(f'call {calls} to {func.__name__}')
        return func(*args, **kwargs)
    return onCall

def timer(label='', trace=True):
    def onDecorator(func):
        def onCall(*args, **kwargs):
            start = time.perf_counter()
            result = func(*args, **kwargs)
            elapsed = time.perf_counter() - start
            onCall.alltime += elapsed
            if trace:
                funcname, alltime = func.__name__, onCall.alltime
                print(f'{label}{funcname}: {elapsed:.5f}, {alltime:.5f}')
            return result
        onCall.alltime = 0
        return onCall
    return onDecorator
```

As we learned in [Chapter 39](#), to use these decorators manually, we simply import them from the module and decorate each function we wish to augment. While this suffices for one-off augmentations, it requires us to add decoration syntax before *each* method we wish to trace or time and to later remove that syntax when we no longer desire the extensions (or use the `-O` trick we'll skip here). If we want to trace or time *every* method of a class, this can become tedious—and may not be possible at all in more dynamic contexts that depend upon runtime parameters.

To do better, [Example 40-9](#) uses a metaclass to add a decorator to *each* of a class's methods automatically. Because the metaclass controls decoration, it can predicate decoration on runtime checks. As a bonus, it can be used for *any* decoration: the decorator to apply to methods is passed as a top-level argument, and hence is allowed to vary per class.

Example 40-9. decoall_meta.py

"Apply any decorator to all methods of a class, with a metaclass"

```
from types import FunctionType
from decorators import tracer, timer

def decorateAll(decorator):
    class MetaDecorate(type):
        def __new__(meta, classname, supers, classdict):
            for attr, attrval in classdict.items():
                if type(attrval) is FunctionType:
                    classdict[attr] = decorator(attrval)
            return type.__new__(meta, classname, supers, classdict)
    return MetaDecorate

class Person(metaclass=decorateAll(tracer)):    # Use a metaclass
    def __init__(self, name, pay):            # Pass any function decorator
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

def tester(aPerson):
    sue = aPerson('Sue Jones', 100_000)
    bob = aPerson('Bob Smith', 50_000)
    print(f'{sue.name=}, {bob.name=}')
    sue.giveRaise(.10)
    print(f'{sue.pay=:.2f}')
    print('Last names:', sue.lastName(), bob.lastName())

if __name__ == '__main__': tester(Person)
```

When this code is run as is, its output traces calls to every method of the client class because every method has been automatically decorated by the metaclass:

```
$ python3 decoall_meta.py
call 1 to __init__
call 2 to __init__
sue.name='Sue Jones', bob.name='Bob Smith'
call 1 to giveRaise
sue.pay=110,000.00
call 1 to lastName
call 2 to lastName
Last names: Jones Smith
```

Really, this result reflects a *combination* of decorator and metaclass—the metaclass automatically applies the function decorator to every method at class creation time, and the function decorator automatically intercepts method calls in order to print the trace messages in this output. The combo “just works,” thanks to the generality of both tools.

To apply a *different* decorator to the methods, simply replace the decorator name in the class header line. To use the timer function decorator shown earlier, for example, use either of the last two header lines in the following for our Person class—the first accepts the timer’s default arguments, and the second specifies label text (though methods may run too fast to register runtimes as is: add `time.sleep(seconds)` calls to pause for a better time):

```

class Person(metaclass=decorateAll(tracer)):           # Apply tracer
class Person(metaclass=decorateAll(timer())):         # Apply timer, defaults
class Person(metaclass=decorateAll(timer(label='**'))): # Decorator arguments

```

And as you might expect by now, class decorators intersect with metaclasses here, too. [Example 40-10](#) replaces the preceding example's metaclass with a class decorator. Really, it uses a class decorator that applies a function decorator to each method of a decorated class. Python's decorators naturally support arbitrary nesting and combinations.

Example 40-10. *decoall_deco.py*

"Apply any decorator to all methods of a class, with a decorator"

```

from types import FunctionType
from decoall_meta import tester
from decorators import tracer, timer

def decorateAll(decorator):
    def DecoDecorate(aClass):
        for attr, attrval in aClass.__dict__.items():
            if type(attrval) is FunctionType:
                setattr(aClass, attr, decorator(attrval))    # Not __dict__
        return aClass
    return DecoDecorate

@decorateAll(tracer)                                     # Use class deco, pass any func deco
class Person:                                           # Applies func decorator to methods
    def __init__(self, name, pay):                     # Person = decorateAll(..)(Person)
        self.name = name                             # Person = DecoDecorate(Person)
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

if __name__ == '__main__': tester(Person)

```

When this code is run, the class decorator applies the tracer function decorator to every method, which in turn produces a trace message on calls (the output is the same as that of the preceding metaclass version of this example):

```

$ python3 decoall_deco.py
...same as Example 40-9...

```

Much as before, we simply mod the @ decorator line to apply a different decorator or provide different arguments for it. Test the following on your own to see the effect (and again add sleep calls as needed to boost times):

```

@decorateAll(tracer)           # Apply tracer
@decorateAll(timer())          # Apply timer, defaults
@decorateAll(timer(label='**')) # Decorator arguments
@decorateAll(timer(label='**', trace=0)) # More decorator arguments

```

Notice that this class decorator returns the augmented class, not a proxy wrapper. As for the metaclass version, this retains the type of the original class—an instance of `Person` is still an instance of `Person`. This may matter when type testing is used. The class's *methods* are not their original functions because they are rebound to decorators, but this is likely less important in practice, and it's true in the metaclass alternative as well.

So far, what we've seen of metaclasses makes them seem largely redundant with decorators—but we have not yet seen all there is to see. As teased earlier, metaclasses may also provide behavior to their instance classes by defining *methods*, which have no direct counterpart in decorators. These methods, however, come with a twist that limits their scope. To understand both metaclasses' methods and their limiting twist, though, we first have to factor metaclasses into Python attribute resolution, a.k.a. *inheritance*, at large. The next section takes us down this prerequisite path.

Inheritance: The Finale

Because metaclasses are coded in similar ways to inheritance superclasses, their scope can be confusing at first glance. In short, there are really two class trees searched by Python inheritance—a *primary* tree formed by a class and that class's superclasses, along with a *secondary* tree formed by a class's metaclass and that metaclass's superclasses. The secondary tree is also called the “type” tree because it stems from type. In more detail, here is how this pans out:

Metaclasses inherit from the type class (usually)

Although they have a special role, metaclasses are coded with normal `class` statements and follow the usual OOP model in Python. As subclasses of `type`, they can redefine the `type` object's methods to customizing classes as needed. Per the prior section, metaclasses often redefine the `type` class's `__new__` and `__init__` to intercept class creation and initialization. Metaclasses can also define methods for their instance classes and may be simple functions or other callables that return arbitrary objects.

*Metaclass attributes **are not** acquired by class instances*

Metaclass declarations specify an *instance* relationship, which is not quite the same as superclass inheritance. Behavior defined in a metaclass applies to the classes made from it but *not* to these classes' own *nonclass* instances. Inheritance for a nonclass instance searches only the instance and the *primary* tree formed by its class and that class's superclasses; the secondary metaclass tree is never included in this search. Hence, nonclass instances get behavior from classes and superclasses but not from metaclasses.

*Metaclass attributes **are** acquired by classes as a fallback*

By contrast, classes *do* acquire methods of their metaclasses by virtue of the *instance* relationship. Metaclasses define a separate inheritance tree, which is a source of class behavior that processes classes themselves. For classes only, inheritance first searches the *primary* tree formed by the class and its superclasses and then falls back on the *secondary* tree formed by the class's metaclass and *its* superclasses as a separate search. When a name is available to a class in *both* a metaclass and a superclass, the superclass version is used.

Metaclass declarations are also inherited by subclasses

The `metaclass=M` declaration in a user-defined class is also *inherited* by the class's normal subclasses in much the same way that superclass links are inherited by subclasses. Thus, the metaclass will run for the construction of each class that inherits this specification in a superclass inheritance chain.

This model's conflation of classes and metaclasses can make terminology challenging, but it may be easier to understand in code than in prose. To illustrate the preceding points, consider the code in [Example 40-11](#).

Example 40-11. metainstance.py

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):      # Redefine type method
        print('In Meta.new:', classname)
        return type.__new__(meta, classname, supers, classdict)
    def meth3(self):
        return 'three!'

class Super(metaclass=Meta):                             # Metaclass inherited by subs too
    def meth2(self):                                     # Meta run twice for two classes
        return 'two!'

class Sub(Super):                                        # Superclass: inheritance versus instance
    def meth1(self):                                    # Classes inherit from superclasses
        return 'one!'                                  # But not from metaclasses for instance access
```

When this file's code is run (as a script or module), the metaclass handles construction of *both* client classes. When those classes are later used, nonclass *instances* inherit class attributes but *not* metaclass attributes:

```
>>> from metainstance import *                          # Runs class statements: metaclass run twice!
In Meta.new: Super
In Meta.new: Sub

>>> X = Sub()                                           # Nonclass instance of user-defined class
>>> X.meth1()                                           # Inherited from Sub
'one!'
>>> X.meth2()                                           # Inherited from Super
'two!'
>>> X.meth3()                                           # Not inherited from metaclass!
AttributeError: 'Sub' object has no attribute 'meth3'. Did you mean: 'meth1'?
```

By contrast, *classes* both inherit names from their superclasses and acquire names from their metaclass—whose linkage in this example is *itself* inherited from a superclass by Sub:

```
>>> Sub.meth1(X)                                       # Own method
'one!'
>>> Sub.meth2(X)                                       # Inherited from Super
'two!'
>>> Sub.meth3()                                       # Acquired from metaclass
'three!'
>>> Sub.meth3(X)                                       # Not a normal instance method!
TypeError: Meta.meth3() takes 1 positional argument but 2 were given
```

Notice how the last of the preceding calls fails when we pass in an instance because the name resolves to a *metaclass method*, not a normal instance method. In fact, both the *source* of a name and the *object* through which you fetch it matter here. Methods acquired from metaclasses are bound to the subject *class*, while methods from normal classes are plain *functions* when fetched through the class but bound with an *instance* when fetched through the instance:

```
>>> Sub.meth3
<bound method Meta.meth3 of <class 'metainstance.Sub'>>
>>> Sub.meth2
<function Super.meth2 at 0x1085f7d80>
>>> X.meth2
<bound method Super.meth2 of <metainstance.Sub object at 0x108456420>>
```

We studied the last two of these cases before in [Chapter 31](#)'s bound-method coverage. The first case is reminiscent of [Chapter 32](#)'s *class methods* but is technically new here. We'll explore class methods in

more detail later. First, though, to understand why metaclass methods aren't available to normal instances, we need to clarify the metaclass/superclass distinction further.

Metaclass Versus Superclass

In even simpler terms, watch what happens in the following: as an *instance* of the *M* metaclass type, class *C* acquires *M*'s attribute, but this attribute is not made available for inheritance by *C*'s own instance *I*—the acquisition of names by metaclass instances is *distinct* from the normal inheritance used for non-class instances:

```
>>> class M(type): attr = 1
>>> class C(metaclass=M): pass          # C is meta instance and acquires meta attr
>>> I = C()                             # I inherits from class but not meta!
>>> C.attr
1
>>> I.attr
AttributeError: 'C' object has no attribute 'attr'
>>> 'attr' in C.__dict__, 'attr' in M.__dict__
(False, True)
```

By contrast, if *M* morphs from metaclass to superclass, then names in superclass *S* become available to later instances of *C* by *inheritance*—that is, by searching the `__dict__` attribute dictionaries of objects in the MRO of *I*'s class as usual, much like the `mapattrs` example we coded back in “[Example: Mapping Attributes to Inheritance Sources](#)” on page 811 (a nonclass-instance-only tool):

```
>>> class S: attr = 1
>>> class C(S): pass                    # C is type instance and inherits from supers
>>> I = C()                             # I inherits from class and supers
>>> C.attr
1
>>> I.attr
1
>>> 'attr' in C.__dict__, 'attr' in S.__dict__
(False, True)
```

This is why metaclasses often do their work by manipulating a new class's namespace dictionary if they wish to influence the behavior of later instance objects—instances will see names in their *class* but not its *metaclass*. Watch what happens, though, if the same name is available in *both* attribute sources—the *inheritance* name in the primary superclass tree is used instead of the *instance* acquisition name in the secondary metaclass tree:

```
>>> class M(type): attr = 1
>>> class S: attr = 2
>>> class C(S, metaclass=M): pass        # Supers have precedence over metas
>>> I = C()                             # Classes search supers before metas
>>> C.attr, I.attr                       # Instances search only supers
(2, 2)
>>> 'attr' in C.__dict__, 'attr' in S.__dict__, 'attr' in M.__dict__
(False, True, True)
```

This is true regardless of the relative height of the inheritance and instance sources—superclass inheritance always beats metaclass acquisition because primary trees are searched before secondary trees:

```
>>> class M(type): attr = 1
>>> class S2: attr = 2
>>> class S1(S2): pass
>>> class C(S1, metaclass=M): pass        # Super two levels above meta: still wins
>>> I = C()
```



```
>>> C.attr, I.attr
(2, 2)
```

In fact, classes acquire metaclass attributes through their `__class__` link, in the same way that nonclass instances inherit from classes through their `__class__`—which makes sense, given that classes are also instances of metaclasses. The chief distinction is that instance inheritance does not also follow a class's `__class__` but instead restricts its scope to the `__dict__` of each class in the superclass tree, following `__bases__` along the way:

```
>>> I.__class__          # Followed by inheritance: instance's class
<class '__main__.C'>
>>> C.__bases__          # Followed by inheritance: class's supers
(<class '__main__.S1'>,)
>>> C.__class__          # Followed by instance acquisition: metaclass
<class '__main__.M'>
```

Really, though, Python checks the `__dict__` of each class on the class's *MRO* before falling back on doing the same for its metaclass—and runs the second of these steps only for fetches run on classes, not nonclass instances:

```
>>> [x.__name__ for x in C.__mro__]    # See Chapter 32 for all things MRO
['C', 'S1', 'S2', 'object']
>>> [x.__name__ for x in M.__mro__]    # Primary/secondary trees: class/meta
['M', 'type', 'object']
```

These two MROs for class and metaclass are simply the flattened versions of what we called the *primary* and *secondary* class trees earlier. Nonclass-instance inheritance searches just the first, but class inheritance searches both if needed. In fact, metaclasses work in much the same way, as the next section explains.

Metaclass Inheritance

As it turns out, instance inheritance works in similar ways, whether the “instance” is a nonclass instance created from a class or a class created from a metaclass derived from `type`. While classes straddle the primary and secondary trees, both trees use the same mechanism to look up names. This yields a single attribute search procedure spanning two trees, which fosters the parallel notion of *metaclass* inheritance hierarchies.

The following demos this conceptual merger. In it, instance `I` inherits from all its classes; class `C` inherits from both superclasses and metaclasses; and metaclass `M1` inherits from higher metaclasses:

```
>>> class M2(type): attr4 = 4          # Metaclass inheritance tree
>>> class M1(M2): attr3 = 3           # Gets __bases__, __class__, __mro__

>>> class S: attr2 = 2                # Superclass inheritance tree
>>> class C(S, metaclass=M1): attr1 = 1 # Gets __bases__, __class__, __mro__

>>> I = C()                          # I gets __class__ but not others
>>> I.attr1, I.attr2                  # Instance inherits from super tree
(1, 2)
>>> C.attr1, C.attr2, C.attr3, C.attr4 # Class gets names from both trees!
(1, 2, 3, 4)
>>> M1.attr3, M1.attr4                # Metaclass inherits names too!
(3, 4)
>>> I.attr3
AttributeError: 'C' object has no attribute 'attr3'. Did you mean: 'attr1'?
```

The failure at the end of this listing is pivotal. Both inheritance paths—class and metaclass—employ the same links to build class MROs scanned by inheritance. But this is not applied *transitively*—instances do not inherit their class’s metaclass names, though they may request them explicitly:

```
>>> I.__class__          # Links used at instance with no __bases__
<class '__main__.C'>
>>> C.__bases__
(<class '__main__.S'>,)

>>> C.__class__          # Links used at class after its __bases__
<class '__main__.M1'>
>>> M1.__bases__         # Link also used in metaclass inheritance
(<class '__main__.M2'>,)

>>> I.__class__.__attr4  # Route inheritance to the class's meta tree
4
>>> I.attr4              # Though class's __class__ not followed normally
AttributeError: 'C' object has no attribute 'attr4'. Did you mean: 'attr1'?
```

And as usual, both trees have flattened MROs and instance links actually used by inheritance:

```
>>> M1.__class__          # Metaclasses are classes too
<class 'type'>
>>> [x.__name__ for x in C.__mro__]  # __bases__ tree from I.__class__
['C', 'S', 'object']
>>> [x.__name__ for x in M1.__mro__] # __bases__ tree from C.__class__
['M1', 'M2', 'type', 'object']
```

If you care about metaclasses—or must use code that does—study these examples carefully. In effect, nonclass instances have no `__bases__` to search but follow `__class__` to `__bases__` once, and classes follow `__bases__` before following a single `__class__` to another `__bases__`. Because this is crucial to the meaning of attribute names in Python, the next section begins making it more formal.

Python Inheritance Algorithm: The Simple Version

Now that we know about metaclass acquisition, we’re finally able to formalize the inheritance rules that they augment. Inheritance deploys two distinct but similar lookup routines, both of which are founded on MROs computed from the prior section’s `__bases__` links per [Chapter 32](#). The combination yields the following first-cut definition of Python’s attribute inheritance algorithm run for explicit attribute fetches.

To look up an attribute name:

1. From a nonclass *instance* I, search the instance, then its class, and then all its superclasses, using:
 - a. The `__dict__` of the instance I
 - b. The `__dict__` of all classes on the `__mro__` found at I’s `__class__`, from left to right
2. From a *class* C, search the class, then all its superclasses, and then its metaclasses tree, using:
 - a. The `__dict__` of all classes on the `__mro__` found at C itself, from left to right
 - b. The `__dict__` of all metaclasses on the `__mro__` found at C’s `__class__`, from left to right

3. In rules 1 and 2, also allow for these exceptions covered ahead:
 - Give precedence to *data descriptors* present in step *b* sources
 - Skip step *a* and begin the search at step *b* for *built-in operations*
 - Perform a custom MRO search for a proxied object in super objects

Rules 1 and 2 are applied for normal, explicit attribute fetch only, and there are exceptions for built-ins, descriptors, and super, which we'll clarify in a moment. In addition, a `__getattr__` or `__getattribute__` may also be used for missing or all names, respectively, per [Chapter 38](#), and attribute assignment follows different rules.

The first two rules here, however, are the essentials. They specify the inheritance search of *two* separate trees, which works the same in each tree but spans trees for *class* inheritance alone. The MRO pseudo-code of [Chapter 32](#) summarizes this even more concisely:

```
[I.__dict__] + [x.__dict__ for x in I.__class__.__mro__]  
  
[x.__dict__ for x in C.__mro__] + [x.__dict__ for x in C.__class__.__mro__]
```

In this, the first line is sources searched by rule 1 for inheritance from a nonclass instance *I*—both the instance itself and the flattened version of the *primary* class tree. The second line is rule 2's sources for inheritance from a class *C*—the flattened versions of both the *primary* class tree and the *secondary* meta-class tree.

Put another way, nonclass instances and classes both search a class tree's MRO at their `__class__` as a second step, but classes first search their own MRO's superclass tree instead of a single `__dict__` namespace dictionary. In code, `class` header lines define both of the trees searched by listing superclasses and specify links to secondary trees with `metaclass` keywords. When omitted, superclass defaults to object and metaclass to type.

Most programmers need only be aware of the first of these rules (1) and perhaps the first step of the second (2a). There's an extra acquisition step added for metaclasses (2b), but it's essentially the same as others—a subtle equivalence, to be sure, but metaclass acquisition is not as novel as it may seem. It's just one step of the procedure.

The descriptors deviation

At least, that's the *normal*—and sugarcoated—case. The prior section listed its exceptions separately because they don't matter in most code and complicate the algorithm substantially. First among these, inheritance also has a special-case coupling with [Chapter 38](#)'s attribute descriptors. In short, *data descriptors*—those that define `__set__` methods to intercept assignments—are given precedence, such that their names override other inheritance sources.

This exception ostensibly serves some practical roles. For example, it is used to ensure that the special `__class__` and `__dict__` attributes cannot be redefined by the same names in an instance's own `__dict__`. In the following, these data-descriptor names in the class *override* same names created in the instance's attribute dictionary:

```

>>> class C: pass                                # Inheritance special case...
>>> I = C()                                       # Class data descriptors have precedence
>>> I.__class__, I.__dict__
(<class '__main__.C'>, {})

>>> I.__dict__['book'] = 'lp6e'                  # Dynamic data in the instance
>>> I.__dict__['__class__'] = 'hack'              # Assign keys, not attributes
>>> I.__dict__['__dict__'] = {}

>>> I.book                                         # I.name comes from I.__dict__ as usual
'lp6e'                                           # But I.__class__ and I.__dict__ do not!
>>> I.__class__, I.__dict__
(<class '__main__.C'>, {'book': 'lp6e', '__class__': 'hack', '__dict__': {}})

```

This data-descriptor exception is tested *before* the preceding two inheritance rules as a preliminary step, may be more important to Python implementers than Python programmers, and can be reasonably ignored by most application code in any event—that is, unless *you* code data descriptors of your own, which follow the same inheritance special case:

```

>>> class D:
    def __get__(self, instance, owner): print('D.__get__')
    def __set__(self, instance, value): print('D.__set__')

>>> class C: d = D()                             # Data-descriptor attribute
>>> I = C()
>>> I.d                                           # Inherited data descriptor access
D.__get__
>>> I.d = 1
D.__set__
>>> I.__dict__['d'] = 'hack'                     # Define same name in instance namespace dict
>>> I.d                                           # But doesn't hide data descriptor in class!
D.__get__

```

Conversely, if this descriptor did *not* define a `__set__`, the name in the instance's dictionary *would* hide the name in its class instead, per normal inheritance:

```

>>> class D:
    def __get__(self, instance, owner): print('D.__get__')

>>> class C: d = D()
>>> I = C()
>>> I.d                                           # Inherited nondata descriptor access
D.__get__
>>> I.__dict__['d'] = 'hack'                     # Hides class names per normal inheritance rules
>>> I.d
'hack'

```

In both cases, Python automatically runs the descriptor's `__get__` when it's found by inheritance rather than returning the descriptor object itself—part of the implicit attribute magic we met earlier in the book. The special status afforded to data descriptors, however, also modifies the meaning of attribute *inheritance* and, thus, the meaning of names in your code. The next section's final swing at a formal inheritance algorithm takes this into account.

Python Inheritance Algorithm: The Less Simple Version

With both the data descriptor special case and general descriptor invocation factored in with class and metaclass trees, Python’s formal inheritance algorithm can be stated as follows—a complex procedure which assumes knowledge of descriptors, metaclasses, and MROs but is the final arbiter of attribute names nonetheless.

To look up an attribute name:

1. From a nonclass *instance* I, search the instance, its class, and its superclasses, as follows:
 - a. Search the `__dict__` of all classes on the `__mro__` found at I’s `__class__`
 - b. If a data descriptor was found in step a, call its `__get__` and exit
 - c. Else, return a value in the `__dict__` of the instance I
 - d. Else, call a nondata descriptor or return a value found in step a
2. From a *class* C, search the class, its superclasses, and its metaclasses tree as follows:
 - a. Search the `__dict__` of all metaclasses on the `__mro__` found at C’s `__class__`
 - b. If a data descriptor was found in step a, call its `__get__` and exit
 - c. Else, call any descriptor or return a value in the `__dict__` of a class on C’s own `__mro__`
 - d. Else, call a nondata descriptor or return a value found in step a
3. In rules 1 and 2, *built-in* operations essentially use just step a sources (see ahead)
4. The super built-in performs a custom MRO search for a proxied object (see ahead)

Some fine print here: in this procedure, items are attempted in sequence as numbered, and Python runs at most *one* (for instances) or *two* (for classes) MRO searches per name lookup—the first appearance of a name in a given MRO wins, regardless of its kind. Because each MRO (a.k.a. `__mro__`) is a flattened representation of a class tree with duplicates removed, you can also think of these as one or two *tree* searches.

In addition, the implied object superclass provides some defaults at the top of every class and metaclass tree (that is, at the end of every MRO). And beyond all this, method `__getattr__` may be run if defined when an attribute is not found, and method `__getattribute__` may be run for every attribute fetch, though they are special-case extensions to the name lookup model (really, the latter replaces inheritance for the defining class’s instances, and can trigger the former with an attribute exception). See [Chapter 38](#) for more on these tools as well as descriptors.

Also, note here again that this algorithm’s first two steps apply only to *normal* and *explicit* attribute *fetch*. The rules for attribute *assignment* vary; the *implicit* lookup of method names for *built-ins* doesn’t follow these rules in full; and the *proxied* lookup of attributes performed by super is entirely custom. The next sections cover these exceptions separately.

The assignment addendum

The prior section defines inheritance in terms of attribute *reference* (a.k.a. fetch or lookup), but parts of it apply to attribute *assignment* as well. As we’ve learned, assignment normally changes attributes in the *subject* object itself, but inheritance is also invoked on assignment to test first for some of [Chapter 38](#)’s attribute management tools, including descriptors, properties, and `__setattr__`. When present, such tools intercept attribute assignment and may implement it arbitrarily.

For example, attribute assignment always invokes a `__setattr__` if present, much as a `__getattr__` is run for all references. Moreover, a *data descriptor* with a `__set__` method is acquired from a class by inheritance using the MRO and has precedence over the normal storage model. In terms of the prior section's rules:

- When applied to an *instance*, attribute assignments essentially follow steps *a* through *c* of rule 1, searching the instance's superclass tree, though step *b* calls `__set__` instead of `__get__`, and step *c* stops and stores in the instance instead of attempting a fetch.
- When applied to a *class*, attribute assignments run the same procedure on the class's metaclass tree: roughly the same as rule 2, but step *c* stops and stores in the class.

Because descriptors are also the basis for other advanced attribute tools such as *properties* and *slots*, this inheritance precheck on assignment is utilized in multiple contexts. The net effect is that descriptors are treated as an inheritance special case for *both* reference and assignment.

The super supplement

Even for attribute *reference*, there are two special cases that are exempt from inheritance's normal rules. For one, the super built-in function we studied in [Chapter 32](#) does not use normal inheritance.

As we learned, for the proxy objects returned by `super`, attributes are resolved by a special context-sensitive scan of a limited *tail* portion of a *different* object's MRO. This custom scan searches the MRO of an implicit instance's class, choosing the first descriptor or value found in a class *following* the class containing the `super` call. This scan is used *instead* of running full inheritance—which is used on the `super` object itself only if this special-case scan fails.

See [Chapter 32](#) for more coverage of `super`. While this built-in may be convenient in some simple roles, it comes with substantial complexity in others and adds a convoluting footnote to inheritance itself.

The built-ins bifurcation

As we've also learned, other *built-ins* don't follow inheritance's normal rules either. Instances and classes may both be skipped for the *implicit* method-name fetches of built-in operations, as a special case that differs from *explicit* name references. Because this is a *context-specific* divergence, it's easier to demonstrate in code than to weave it into a single algorithm. In the following, `str` is the built-in, `__str__` is its explicit-name equivalent, and the *nonclass* instance is inconsistently skipped by the built-in only:

```
>>> class C:                                # Inheritance special case...
    attr = 1                                # Built-ins skip a step
    def __str__(self): return('class')

>>> I = C()
>>> I.__str__(), str(I)                      # Both from class if not in instance
('class', 'class')

>>> I.__str__ = lambda: 'instance'
>>> I.__str__(), str(I)                      # Explicit=>instance, built-in=>class!
('instance', 'class')

>>> I.attr                                  # Asymmetric with normal or explicit names
1
>>> I.attr = 2; I.attr
2
```

As you may expect by now, the same holds true for *classes*—explicit names start at the class, but built-ins start at the class’s class—which is its *metaclass*, and defaults to `type`, which provides access to an implicit default:

```
>>> class D(type):
    def __str__(self): return('D class')
>>> class C(D):
    pass
>>> C.__str__(C), str(C)                                # Explicit=>super, built-in=>metaclass!
('D class', "<class '___main__.C'>")

>>> class C(D):
    def __str__(self): return('C class')
>>> C.__str__(C), str(C)                                # Explicit=>class, built-in=>metaclass!
('C class', "<class '___main__.C'>")

>>> class C(metaclass=D):
    def __str__(self): return('C class')
>>> C.__str__(C), str(C)                                # Built-in=>user-defined metaclass
('C class', 'D class')
```

In fact, it can sometimes be nontrivial to know *where* a name comes from in this model since all classes in both trees also inherit from `object`—including the default `type` metaclass. In the following’s explicit call, `C` gets a default `__str__` from `object` instead of the metaclass, per the first source of class inheritance (the class’s own MRO, which is the primary tree); by contrast, the `str` built-in skips ahead to the metaclass as before:

```
>>> class C(metaclass=D):
    pass
>>> C.__str__(C), str(C)                                # Explicit=>object, built-in=>metaclass
("<class '___main__.C'>", 'D class')

>>> C.__str__
<slot wrapper '__str__' of 'object' objects>

>>> for k in (C, C.__class__, type):
    print([x.__name__ for x in k.__mro__])

['C', 'object']
['D', 'type', 'object']
['type', 'object']
```

This is why we’ve gone to such great lengths to root out the full specs of inheritance. While some code may never need to care about all its many plot twists, attribute inheritance is clearly a convoluted business in Python, and uncertainty is not generally compatible with engineering.

The Inheritance Wrap-Up

And with all those details in hand, you finally have the complete Python inheritance story—or at least as much as we can cover in this text. It’s a tangled tale that today spans instances, classes, superclasses, metaclasses, descriptors, `super`, built-ins, and MROs, and all for the sake of looking up a simple attribute name.

Some practical needs warrant exceptions, of course, but you should carefully consider the implications of an object-oriented language that applies inheritance—its *foundational* operation—in such a *labyrinthian* fashion. At a minimum, this should underscore the importance of keeping your own code simple

to avoid making it dependent on the darker corners of such convoluted rules. As always, your code’s users and maintainers will be glad you did.

For more fidelity on this story, see Python’s internal implementation of inheritance—a low-level but complete saga chronicled today in its files *object.c* and *typeobject.c*, the former for normal instances and the latter for classes. Delving into internals shouldn’t be required to use Python, but it’s the ultimate and sometimes sole source of truth in a complex and perpetually changing system. This is especially true in boundary cases born of accrued exceptions that raise the bar for learners and users, a downside we’ll revisit briefly in the next and closing chapter.

For now, let’s move on to one last bit of metaclass “magic”—its methods, which rely on its inheritance offshoot.

Metaclass Methods

Now that we have a handle on the way that metaclasses modify the inheritance of names, we can finally turn to their methods with full clarity. In short, *methods* in metaclasses are inherited by and process their instance *classes*—instead of the nonclass instances that classes make.

This makes metaclass methods similar in form and function to the *class methods* we studied in [Chapter 32](#), though their class-focused behavior is automatic. Moreover, they are available only to classes due to the last section’s inheritance rules—the limiting “twist” alluded to earlier. Metaclass methods also have no direct analog in class decorators, though decorators’ freedom to return arbitrary objects makes nearly anything possible with imagination.

To demo the basics, the following’s metaclass defines a method made available to its class’s instances by metaclass acquisition (i.e., by inheritance from the secondary metaclass tree):

```
>>> class M(type):
    def z(cls): print('M.z', cls)           # A metaclass: instances=classes
    def y(cls): print('M.y', cls)         # y is overridden by instance C

>>> class C(metaclass=M):
    def y(self): print('C.y', self)        # A simple class: nonclass instances
    def x(self): print('C.x', self)        # Namespace dict holds x and y
```

Methods fetched from the class are plain functions as usual, but those from a metaclass are automatically bound to the class from which they were fetched, as we saw in an earlier example:

```
>>> C.x
<function C.x at 0x10eaf4b80>
>>> C.y                                     # x and y defined in class itself
<function C.y at 0x10eaf4ae0>
>>> C.z                                     # z acquired from metaclass
<bound method M.z of <class '__main__.C'>>
>>> C.z()                                   # Metaclass method call: gets class
M.z <class '__main__.C'>
```

As we’ve also seen, methods fetched through a nonclass instance are bound to the instance unless *class* method or *staticmethod* are used, though such instances are moot here because they do not inherit metaclass names:

```
>>> I = C()
>>> I.x()                                   # Instance method calls: get inst
C.x <__main__.C object at 0x10e9d1400>
>>> I.y()
C.y <__main__.C object at 0x10e9d1400>
>>> I.z()                                   # Instance doesn't see meta names
```



```

AttributeError: 'C' object has no attribute 'z'
>>> I.x
<bound method C.x of <__main__.C object at 0x10e9d1400>>

```

The only real new things about metaclass methods is that they are inherited only by classes and are bound to a class automatically. The latter half of this is explored in the next section.

Metaclass Methods Versus Class Methods

Though they differ in inheritance visibility, metaclass methods are designed to manage class-level data, just like the *class methods* we studied in [Chapter 32](#). In fact, their roles can overlap—much as metaclasses can in general with class decorators—but metaclass methods are not accessible except through the class and do not require an explicit `classmethod` class-level declaration in order to be bound with the class.

In other words, metaclass methods can be thought of as *implicit* class methods, with limited *visibility*:

```

>>> class M(type):
    def a(cls):
        cls.x = cls.y + cls.z
        # Metaclass method: gets class

>>> class C(metaclass=M):
    y, z = 11, 22
    @classmethod
    def b(cls):
        return cls.x
        # Class method: gets class

>>> C.a()
# Call metaclass method; visible to class only
>>> C.x
# Creates class data on C, accessible to normal instances
33

>>> I = C()
>>> I.x, I.y, I.z
(33, 11, 22)

>>> I.b()
# Class method: sends class, not instance; visible to instance
33
>>> I.a()
# Metaclass methods: accessible through class only
AttributeError: 'C' object has no attribute 'a'

```

This yields two very different ways to create class methods whose asymmetry is left as suggested pondering here.



Plus one more: Python 3.6 added an operator-overloading method named `__init_subclass__`. This method is called whenever the containing class is *sub-classed*, and it receives a single argument: the new subclass object. It provides yet another way to manage classes, though this method is nowhere near as general as class decorators or metaclasses: it's run only for a class's own subclasses and only when those subclasses are created. Similar to metaclass methods, this method is also implicitly converted to a *class method*. Because we just can't get enough of those implicit hooks!

Operator Overloading in Metaclass Methods

Just in case your head is not yet spinning, metaclasses, just like normal classes, may also employ operator overloading to make built-in operations applicable to their instance classes. The `__getitem__` indexing method in the following metaclass, for example, is a metaclass method designed to process *classes* themselves—the classes that are instances of the metaclass, not those classes' own nonclass instances:

```
>>> class M(type):
    def __getitem__(cls, i):
        return cls.data[i]
    # Meta method for processing classes:
    # Built-ins skip class, use meta
    # Explicit names search class + meta
>>> class C(metaclass=M):
    data = 'hack'
    # Data descriptors in meta used first

>>> C[0]
'h'
# Metaclass instance names: visible to class only
>>> C.__getitem__
<bound method M.__getitem__ of <class '__main__.C'>>

>>> I = C()
>>> I.data, C.data
('hack', 'hack')
# Normal inheritance names: visible to instance and class
>>> I[0]
TypeError: 'C' object is not subscriptable
```

And if that's not abstruse enough, when both class and metaclass overload the *same* operator, the former applies to classes and the latter to nonclass instances:

```
>>> class C(metaclass=M):
    data = 'hack'
    def __getitem__(self, i):
        return self.data[i]

>>> I = C()
>>> I.data = 'code'
>>> C[0], I[0]
('h', 'c')
# C's [] from metaclass, I's [] from class
```

All of which leads us to the closing compare-and-contrast of the next section.

Metaclass Methods Versus Instance Methods

The inescapable conclusion of this technical novel is that metaclass methods provide a separate—and arguably redundant—way to code object behavior with classes in Python. As we've learned, the usual way to implement objects is with classes and their nonclass instances. Here's the *normal* case—with class data, constructor, regular method, and operator overloading available to instances and subclasses:

```
>>> class Employee:
    rate = 50
    def __init__(self, name, hours):
        self.name = name
        self.hours = hours
    def pay(self):
        print(f'💰 {self.name} ➡️ ${self.rate * self.hours:,.2f}')
    def __iadd__(self, hours):
        self.hours += hours
        return self
    # A "normal" class
    # With shared class data
    # Plus instance data and methods
```

As usual, we make an instance of a normal class like this by calling it with constructor arguments:

```
>>> pat = Employee('Pat', 2_000)           # A "normal" nonclass instance
>>> pat.pay()                               # Methods inherited from class
💰 Pat ➡ $100,000.00
>>> pat += 1_000                             # Updates instance data
>>> pat.pay()
💰 Pat ➡ $150,000.00
```

And to make more instances, we simply call the class again:

```
>>> pat2 = Employee('Pat2', 1_000)          # Another "normal" instance
>>> pat2.pay()
💰 Pat2 ➡ $50,000.00
>>> pat2
<__main__.Employee object at 0x10cf5c680>
```

The equivalent *metaclass* can also provide data, regular methods, and operator overloading. But per-instance constructors don't quite apply; methods receive and process a class, not instances of it; and this behavior is for subclasses and instance classes in the secondary “type” tree only:

```
>>> class MetaEmployee(type):                # A metaclass class
    rate = 50                                # With metaclass data and methods
    def pay(cls):
        print(f'💰 {cls.name} ➡ ${cls.rate * cls.hours:,.2f}')
    def __iadd__(cls, hours):
        cls.hours += hours
        return cls
```

Because metaclass instances are classes, we define a new *class* per instance instead of running a call and code instance data as class attributes, though all the behavior defined in the metaclass applies to its instance classes:

```
>>> class pat(metaclass=MetaEmployee):       # A metaclass instance: class
    name = 'Pat'                             # With class data per instance
    hours = 2_000

>>> pat.pay()                               # Methods inherited from metaclass
💰 Pat ➡ $100,000.00
>>> pat += 1_000                             # Updates class data
>>> pat.pay()
💰 Pat ➡ $150,000.00
```

We haven't made any normal instances here—just a *class* that nevertheless serves as an information record with both data and behavior methods. Coding differences aside, the main functional divergence in the metaclass approach is that *nonclass* instances created from such a class don't inherit any of the behavior defined in a metaclass:

```
>>> pat2 = pat()                            # A "normal" instance of class pat
>>> pat2.pay()                              # Metaclass behavior not available
AttributeError: 'pat' object has no attribute 'pay'
```

To make additional instances in the metaclass world, we must instead code additional classes:

```
>>> class pat2(metaclass=MetaEmployee):      # Metaclass instances are classes
    name = 'Pat2'
    hours = 1_000

>>> pat2.pay()                              # With all metaclass data+methods
💰 Pat2 ➡ $50,000.00
```

Calling the metaclass with *no* arguments doesn't work at all because it makes a *class*, not an instance of a class—though calling the metaclass with full type arguments *does* work because it's equivalent to running a `class` statement (and yes, the fact that metaclasses are classes, too, makes some of this paragraph's logic circular, but that's just how Python works: metaclasses are special-cased for metaclass roles):

```
>>> pat2 = MetaEmployee('pat2', (), dict(name='Pat2', hours=1_000))
>>> pat2.pay()
💰 Pat2 ➡ $50,000.00
>>> pat2
<class '__main__.pat2'>
```

In sum, metaclasses allow us to implement classes that work much like the nonclass instances we've used throughout this book. As to *why* you'd want to swap one kind of instance for another with identical functionality but different coding, we'll have to defer to other resources to justify the *redundancy*. Given the many convolutions that metaclasses bring to the table, it's difficult not to see this as complexity for all in the name of rare and narrow roles. While that has been a recurring theme in both Python and this book, it's time to wrap up and move on to the conclusion.

Chapter Summary

In this chapter, we studied metaclasses, explored examples of them in action, and formalized the rules of inheritance that they extend. Along the way, we also saw how the roles of class decorators and metaclasses often intersect: because both run at the conclusion of a `class` statement, they can sometimes be used interchangeably. We also learned how metaclass methods define behavior for classes much like the class methods we met earlier, but are limited in scope to the secondary inheritance tree searched for classes.

Since this chapter covered an advanced topic, we'll work through just a few quiz questions to review the basics (candidly, if you've made it this far in a chapter on metaclasses, you probably already deserve extra credit!). Because this is the last part of the book, we'll also forgo the end-of-part exercises. Be sure to see the appendixes that follow for Python platform pointers and the solutions to the prior parts' exercises; the last of these includes a sampling of short application-level programs for self-study.

Once you finish the quiz, you've officially reached the end of this book's technical material. The next and final chapter offers some brief thoughts about Python to wrap up the book at large and closes with some fun. We'll regroup there in the Learning Python benediction after you work through this final quiz.

Test Your Knowledge: Quiz

1. What is a metaclass?
2. How do you declare the metaclass of a class?
3. How do class decorators overlap with metaclasses for managing classes?

Test Your Knowledge: Answers

1. A metaclass is a class used to create a class. Classes are instances of the type class by default. Metaclasses are usually subclasses of the type class, which customize classes. They may redefine class-creation protocol methods like `__new__` and `__init__` to customize the class creation call issued at the end of a `class` statement. They may also define data and methods that provide behavior for their instance classes, but these are inherited only by classes, not their nonclass instances. Metaclasses can also be coded in other ways—as simple functions, for example—but they are responsible for making and returning an object for the new class. Finally, metaclasses constitute a secondary pathway for inheritance search used for classes alone; this allows classes to ape normal instance behavior, though it bifurcates the class story for relatively rare and narrow roles.
2. Use a keyword argument in the `class` header line: `class C(metaclass=M)`. The `class` header line can also name normal superclasses before the `metaclass` keyword argument; these superclasses are searched before metaclasses for inheritance run from a class.
3. Because both are automatically triggered at the end of a `class` statement, class decorators and metaclasses can both be used to manage classes. Decorators rebind a class name to a callable's result, and metaclasses route class creation through a callable, but both hooks can be used for similar purposes. To manage classes, decorators simply augment and return the original class objects. Metaclasses augment a class after or before they create it. As noted, metaclasses can also provide behavior for their instance classes in the form of methods that are not immediately supported by decorators, though the objects returned by decorators can do anything supported by the Python language.

All Good Things

Welcome to the end of the book! Now that you’ve made it this far, this chapter says a few words in closing about Python’s evolution before turning you loose on the software field and then wraps up with a bit of fun.

You’ve now had a chance to see the entire Python language yourself—including some advanced features that may seem at odds with a scripting language meant to be accessible to nonprofessionals. Though many users will understandably accept this as status quo, in an open source project, it’s crucial that some ask the “why” questions too. Ultimately, the trajectory of the Python story—and its true conclusion—is at least in part up to you.

Toward that end, this chapter begins by calling out what may be one of Python’s biggest downsides: its rate of change. This topic is unavoidably subjective, and you should weigh its coverage here on whatever scale you bring to the table.

The Python Tsunami

Twelve years ago, this book warned that Python was growing too convoluted and bloated—and then Python grew a lot more convoluted and bloated. Clearly, this message has not reached those behind the convoluting and bloating.

Even so, this stuff still matters. To parrot the Preface, the last dozen years have hosted the rise of f-string literals, named-assignment expressions, `match` statements, type hinting, async coroutines, dictionary union, star-unpacking proliferation, underscore digit separators, module attribute hooks, exception groups, dictionary-key insertion order, positional-only function arguments, hash-based bytecode files, the `sys.executable` snub, and other superfluous additions, opinionated deprecations, and tangled mutations we’ve met along the way.

Moreover, this *tsunami* of mods simply added to the flood of complexity and redundancy that came before it—including the oddly artificial MRO, the stunningly implicit `super`, and the horrifically convoluted inheritance algorithm of the preceding chapter, which elevates metaclasses and descriptors to prerequisites. The sum of these is an esoteric morass, which obfuscates the fundamental meaning of names in Python.

Most of what should be said about these changes already has been said in this book, but their combined weight qualifies as problematic for a tool that promotes itself as simpler than others. To illustrate,

Table 41-1 updates the prior edition’s accounting of Python’s largely unchecked growth so far—a partial but representative tally.

Table 41-1. A sampling of redundancy and tool explosion in Python

Category	Members
3 major paradigms	Procedural, functional, object-oriented
4 string-formatting tools	% expression, <code>str.format</code> , <code>string.Template</code> , f-strings
4 attribute-accessor tools	properties, descriptors, <code>__getattr__</code> , <code>__getattribute__</code>
2 finalization statements	<code>try/finally</code> , with plus context managers
4 varieties of comprehension	List, set, dictionary, generator
3 class-augmentation options	Manual rebinding, @ decorators, metaclasses
4 kinds of methods	Instance, static, class, metaclass
2 attribute-storage systems	<code>__dict__</code> , <code>__slots__</code>
4 flavors of imports	Module, package, package relative, namespace package
2 superclass-reference tools	Explicit class names, <code>super</code> plus MRO
6 assignment forms	Basic, sequence, multitarget, += augmented, * unpacking, := named
3 types of functions	Basic, <code>yield</code> generator, <code>async</code> coroutine
6 function-argument forms	Basic, <code>name=X</code> , *X, **X, keyword-only, positional-only
2 class-behavior sources	Superclasses, metaclasses
3 multiple-choice tools	<code>if/elif</code> , dictionary indexing, <code>match</code>
4 state-retention options	Classes, closures, function attributes, mutables
2 bytecode storage schemes	timestamp, hashkey
3 name-string importers	<code>exec</code> , <code>__import__</code> , <code>importlib</code>
2 kinds of decorators	Function, class
4 dictionary-merge options	<code>for</code> loops, update method, *D unpacking, union operator
2 exception-handler models	<code>except</code> singles, <code>except*</code> groups
4 statement-aping expressions	<code>if/else</code> , comprehensions, <code>lambda</code> functions, := assignment
8 starred collectors/unpackers	Assignment; function header, call; list, tuple, dict, set literal; <code>match</code>

If you care about Python, you should take a moment to browse this table. It reflects a virtual *explosion* of bifurcation, redundancy, and toolbox size—and 81 concepts that can all be required reading for both newcomers learning the language and experts reusing code written by others. Most of its categories began with just one original member in Python; many were added in part to imitate other languages; and none can be simplified today by pretending that the latest Python is the only Python that matters. Python 3.X now owns the flux in full.

F-strings are a prime example in this category. This book’s prior edition lamented the three redundant string-formatting tools of its time, but this set was subsequently expanded to a colossal four. While f-strings may be deemed a refinement by some, in truth they are a minor variation on a theme that adds yet another topic to the heap. More fundamentally, millions of programmers have written millions of programs using longer-lived options; while new code has the luxury of using new tools, pretending that the past didn’t happen constitutes a break with reality.

Even extensions perhaps more unique often come loaded with surplus complexity. The `match` statement, for example, couldn’t simply provide a potentially useful multiple-choice option. It had to bolt on

the conceptually tortuous and syntactically ad hoc structural pattern matching, which seems an answer to a question that nobody asked.

Nor are additions the only user-unfriendly theme in Python. Its subjective *changes* and *deprecations* are now so common that they must be expected as an implicit cost of using the language. To be clear, your Python code will almost certainly break eventually when you upgrade to a new Python release—and only because a Python core developer’s whim was made mandatory for everyone else.

The Python Sandbox

All of this stems from the fact that Python is, and probably always has been, a constantly morphing *sandbox* of ideas, which prioritizes its developers’ egos over its users’ needs. Playing in a sandbox can be fun, of course, but it’s lousy for the millions of people downstream from its churn. These people are simply trying to write software that’s reliable and durable. Like all engineering endeavors, that works best with a *stable base*, not constantly shifting sand.

As a pathological example, Python’s sandbox model seems to have hit its zenith in *type hinting*—the optional, unused, out-of-place, and embarrassingly academic subdomain we glanced in [Chapter 6](#) but largely omitted here by design. This is unchecked convolution on parade, and leaves Python users to puzzle over the paradox of pointless type declarations in a dynamically typed language. Sadly, it’s also likely to appeal to control freaks.

As we’ve also seen regularly in this book, because the sandbox is oriented toward experts, it inevitably produces tools that assume that you have to already be an expert to use them. Classes and OOP, for example, are required skills for even simple exception handling. Python is not just for its developers, but the *forward knowledge* assumptions of many of its additions embed this message and raise the bar for newcomers unnecessarily and unkindly.

To be fair, it’s not just Python: the entire software field is permeated by a *culture of change* in which churn is an expected constant, and prowess often consists of flaunting the latest and greatest tools even when they are unwarranted. This doesn’t prove intelligence (and often demos its absence), but annual and mandatory mods are now a norm. Whether one breaks your PC, smartphone, or Python code, it’s difficult not to see this as divisive and rude.

The Python Upside

All that being said, it’s also difficult to deny that Python, despite its warts, is still more productive and pleasant to use than other programming languages. If you’ve coded other languages, you know that many come laden with extraneous syntax and rules, which seem to reflect an assumption that programmers cannot be trusted to do their jobs. By sharp contrast, Python’s dynamic typing and innate flexibility make it more ally than obstacle.

Python’s rise in *popularity* seems to attest to this value proposition, though its impetus may be more practical than academic. Today’s larger Python world may naturally be less concerned with the language’s original and perhaps idealistic goals than with solving concrete problems. In the real world that hosts Python popularity, arcane language topics usually take a back seat to libraries, platforms, and schedules—calls that Python has always answered in full.

Moreover, some change and complexity is *warranted* in software. Programming is a substantially challenging task (despite what you may have heard), and computer science is a field still young enough to be excused for some youthful thrashing. For Python specifically, though, complexity and thrashing should be modulated by broad appeal.

In the end, the Python language remains a remarkably expressive tool that still fits both programming tasks and your brain as well as it ever did. Especially if you stick to its tried-and-true parts that propelled Python to the top of the language charts, you'll likely find it an enabling technology that makes coding as much fun as chore.

Prudent engineers, though, would do well to exercise caution when upgrading to the leading edge, and give a pass to the sandbox's annual outflow except when clearly beneficial. Given that this is now an industry-wide requirement, it's hardly cause to dismiss an otherwise useful tool. Bad practice, however, does not justify bad practice.

Closing Thoughts

So there you have it: some observations from the trenches, born of three decades using, teaching, and advocating Python, and grounded in a desire to improve the Python story.

None of these concerns are entirely new. Indeed, the growth of this very book over the years seems a testament to that of Python itself—if not an ironic eulogy to a mission statement that once stressed simplification of programming, and accessibility to both experts and nonprofessionals alike. Judging by language heft alone, that dream seems to have been either neglected over time or abandoned entirely.

But we can do better. A well-established tool like Python can afford to focus more on its users' needs than its changers' hubris. Per the old adage, we simply have to stop fixing what isn't broken. If we can, it will go far toward addressing the concerns of those vetting the language for projects that cannot afford to budget for shifting sands.

More importantly, in an open source project like Python the answers to such questions must be formed anew by each wave of newcomers. Hopefully, the wave you ride in will have as much common sense as fun while plotting Python's future.

Where to Go from Here

And that's a wrap, folks. You've officially reached the end of this book. Now that you know Python inside and out, your next step, should you choose to take it, is to explore the libraries, techniques, and tools available in the application domains in which you will work.

Because Python is so widely used, you'll find ample resources for using it in almost any domain you can think of—from GUIs, the web, and apps, to numeric programming, databases, and system administration. See [Chapter 1](#) and your favorite web browser for pointers to popular tools and topics.

This is where Python starts to become truly fun, but this is also where this book's story ends, and those of other resources begin. Good luck with your journey. And as always: code well!

Encore: Print Your Own Completion Certificate!

And one last thing: in lieu of exercises for this part of the book, [Example 41-1](#) lists a bonus script for you to study and run on your own. A book can't directly provide completion certificates for its readers (and the certificates would be worthless if it could), but it can include an arguably cheesy Python script that does. This one creates a simple book completion certificate in both plain-text and HTML files, and auto-opens them in a web browser or other viewer where supported.

Example 41-1. You-made-it.py

```
"""
Generate a simple class completion certificate: printed to
the console, and saved in auto-opened text and HTML files.
Run from a console, and print saved output files if desired.
Works on all PCs, but may require manual opens on smartphones.
"""

import time, sys, html, os

maxline = 60                                # Text separator lines
browser = True                               # Display in a browser?
saveto = 'Certificate'                       # Output filenames prefix

# Template values
SEPT = '*' * maxline
DATE = time.strftime('%A, %b %d, %Y, %I:%M %p')
NAME = input('Please enter your name: ').strip() or 'An unknown reader'
BOOK = 'Learning Python, 6th Edition'
SITE = 'https://learning-python.com'        # For icon, image, link

# F-string templates work for preset in-code references
textttext = f"""
{SEPT}

★ Official Certificate ★

Date: {DATE}

This certifies that:

\t{NAME}

Has survived the massive tome:

\t{BOOK}

And is now entitled to all privileges thereof, including
the right to proceed on to learning how to develop websites,
desktop GUIs, scientific models, smartphone apps, and
anything else that the future of computing may hold.

--Your humble 🐍 instructor

(Note: void where obtained by skipping ahead.)

{SEPT}
"""

# Interact, setup
for c in 'Congratulations!'.upper() + '🎉' * 3:
    print(c, end=' ')
    sys.stdout.flush()    # Else some shells wait for \n
    time.sleep(0.25)       # Reveal message slowly for fun
print(); time.sleep(3)

# Make text-file version
textto = saveto + '.txt'
fileto = open(textto, 'w', encoding='utf8')
print(textttext, file=fileto)
fileto.close()
```

```

# Start HTML: replace text markers with tags
htmltext = texttext.replace(SEPT, '<div class=cert>', 1)
htmltext = htmltext.replace(SEPT, '</div>')
htmltext = htmltext.replace('★', '<h1 align=center>★&nbsp;  ', 1)
htmltext = htmltext.replace('★', '&nbsp;  ★</h1>')

# Line-by-line mods
linemods = []
for line in htmltext.split('\n'):
    if line == '':
        line = '<p>'
    elif line[:1] == '\t':
        line = f"<i>{'&nbsp;' * 4}{html.escape(line[1:])}</i>" # 3.6+
    linemods.append(line)
htmltext = '\n'.join(linemods)

# Ignorable HTML bits (mind the {{ and }} escapes)
preamble = f'''<!doctype html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<link rel="icon" type="image/x-icon" href="{SITE}/favicon.ico">
<style>
body {{font-family: Arial, Helvetica, sans-serif;}}
.cert {{background-color: cornsilk; padding: 16px; border: medium solid black;}}
</style>
<title>LP6E Completion Certificate</title>
</head>
'''

image, page = 'lp6e-large.jpg', 'about-lp.html'
footer = f'''
<table><tr>
<td><a href="{SITE}/{page}"></a>
<td><a href="{SITE}/{page}" align=center><i>Book support site</i></a>
</tr></table>
'''

# Put it all together
htmltext = f'{preamble}<body bgcolor="#eee">{htmltext}{footer}</body></html>'

# Make HTML-file version
htmlto = saveto + '.html'
fileto = open(htmlto, 'w', encoding='utf8')
print(htmltext, file=fileto)
fileto.close()

# Display text results in console
print(f'[File: {textto}]', end='')
print('\n' * 2, open(textto, encoding='utf8').read())

# Open docs (may also fail silently)
if browser:
    try:
        import webbrowser
        for doc in (textto, htmlto):
            webbrowser.open('file://' + os.path.abspath(doc))
    except Exception:

```

```
print('Unable to auto-open docs: open manually.')

input('Press Enter to close')    # Keep window open if clicked
```

Run this script in a console or other interface on your own, and study its code for a review of some of the ideas we’ve covered in this book. Copy/paste from emedia or fetch it from this book’s examples package as described in the [Preface](#), but ignore its undocumented and out-of-scope HTML bits if you’re not a web developer. You won’t find any descriptors, decorators, metaclasses, or super calls in this code, but it’s typical Python nonetheless:

```
$ python3 You-made-it.py
Please enter your name: Some Body
C O N G R A T U L A T I O N S ! 🎉 🎉 🎉
...etc...
```

This script works in full on PCs (where code might also open files with `os.startfile`, or “open” or “xdg-open” commands in `os.system`), but on smartphones you’ll probably need to open the output files manually in file-explorer apps. When run, it generates the web page captured in the fully gratuitous [Figure 41-1](#). This could be much more grandiose, of course; see the web for pointers to Python support for PDFs and other document tools such as Sphinx surveyed in [Chapter 15](#). But hey—if you’ve made it to the end of this book, you deserve another joke or two.

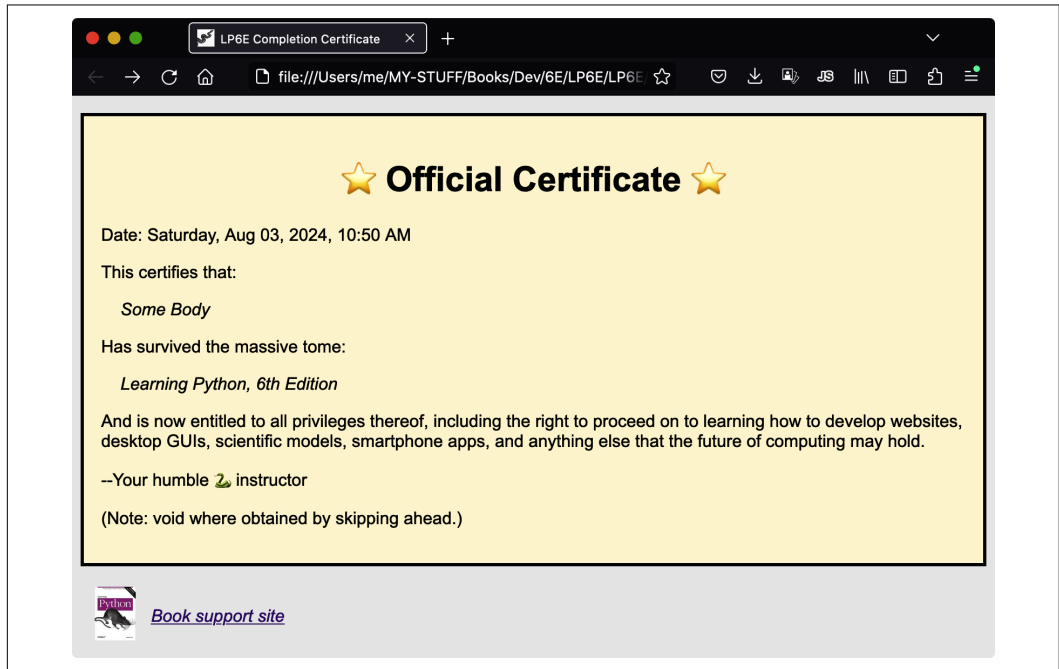


Figure 41-1. HTML doc created and opened by *You-made-it.py*

Appendixes

Platform Usage Tips

One of Python's main strengths is its *portability*: most of the code you'll write in your Python programs will work the same on all computing platforms. This has limits, of course; programs that use Python's portable libraries weather device hops better than others, and platform idiosyncrasies and restrictions can sometimes pose interoperability hurdles that require special handling. But by and large, the Python language is cross-platform by design.

Python's portability means you can run its code on just about every computing device on the planet—from smartphones and tablets to PCs and supercomputers—and each of these systems has unique setup and usage details. While this appendix cannot be an exhaustive user guide for every one of those devices, it provides just enough info to help you prepare to run this book's code examples on your popular platform—or platforms—of choice, including Windows, macOS, Linux, Android, and iOS.

Before we get started, here are two quick content notes up front. First, because you're going to have enough on your plate just learning Python itself, the focus throughout this appendix is on *keeping it simple*. There are many ways to run code, and you may find advanced options useful once you graduate from Python novice to master. Especially when starting out, though, this book recommends walking the easiest path.

Second, a usage appendix like this is unavoidably doomed to grow out of date soon, given the rapid and constant change in the computing world (even the name of macOS, after all, has changed repeatedly on this book's watch!). Hence, please consider this a *snapshot* of the current state and practice, and plan to consult the latest resources if when this story changes. For the present, let's jump right into today's usage options—while they last.

Using Python on Windows

As the current market-share leader for PCs, Windows will undoubtedly play host to many readers' first encounter with Python. Python has been completely usable on this platform since its earliest days, and goes out of its way to smooth Windows' proprietary edges so your code doesn't have to. A well-coded Python program from Unix, for example, often runs unchanged on Windows despite the two platforms' many glaring differences.

Today, there are at least three different ways to use Python on Windows: in Windows itself, in Windows Subsystem for Linux (a.k.a. *WSL* and *WSL2*), and in the third-party *Cygwin* Unix-like environment. We

won't cover Cygwin here because it's not as widely used, and those who may care to use it probably already know how to use it.

WSL—including its newer WSL2 variant—brings Linux to your Windows PC without the hassles of dual-boot installs or separate devices. You get a standard Linux distribution (e.g., Ubuntu) that runs in the Windows UI and avoids some of the trade-offs of classic virtual machines. WSL comes with a command line to edit and run Python code, and WSL2 even runs Linux GUI apps (though they're still marginal at this writing). Because WSL is really Linux, we'll defer to the Linux section ahead for Python setup and usage info, as well as Microsoft's online documentation for details on installing WSL itself.

While WSL may pique some readers' interest, it requires extra setup steps and is not wholly without seams today. For most readers, the easiest way to use Python on Windows is to go *native*—with a Python built to run in Windows directly. Python doesn't come with Windows, but it is easy to install and use there. In short, Python for Windows can be installed by downloading a self-installer or visiting the Microsoft Store, and can be run with a simple Windows command-line interface; a graphical *IDE*—a GUI for editing and launching code; and clicks on program-file entries in Windows File Explorer.

In more detail, the recommended way to *install* Python for Windows begins with a visit to the Downloads page at python.org. There, you'll fetch a Windows self-installer that you'll run to install Python 3.X, along with its IDLE GUI and standard library (including the library's `tkinter` GUI toolkit). **Figure A-1** captures the Python installer in action; allow it to run if Windows asks for permission. You can generally accept all installation defaults, but it's recommended to opt in to both adding Python to your PATH at the start of the install and lifting the Windows path-length limits for filenames at the end.



Installation convolution: Though less common, you can also install Python from the *Microsoft Store*. In fact, typing `python3` in a Windows command line at this writing automatically routes you to the store to run the install—confusingly! If you opt to accept this offer, `python3` will run the store's version after the install, and the Start menu will sprout separate entries for launching this Python and its IDLE (up shortly).

This guide uses and generally recommends the *python.org* install (and its `py` helper) because it's more traditional and may be better suited to general use. That said, the store version ultimately comes from the same source, and either may be used for this book's examples on Windows.

But beware: the store version imposes access restrictions that may matter to you later and should probably relegate it to a secondary option for most readers. You really shouldn't install *both* the store and nonstore Windows Pythons, though, unless you need more drama in your life!

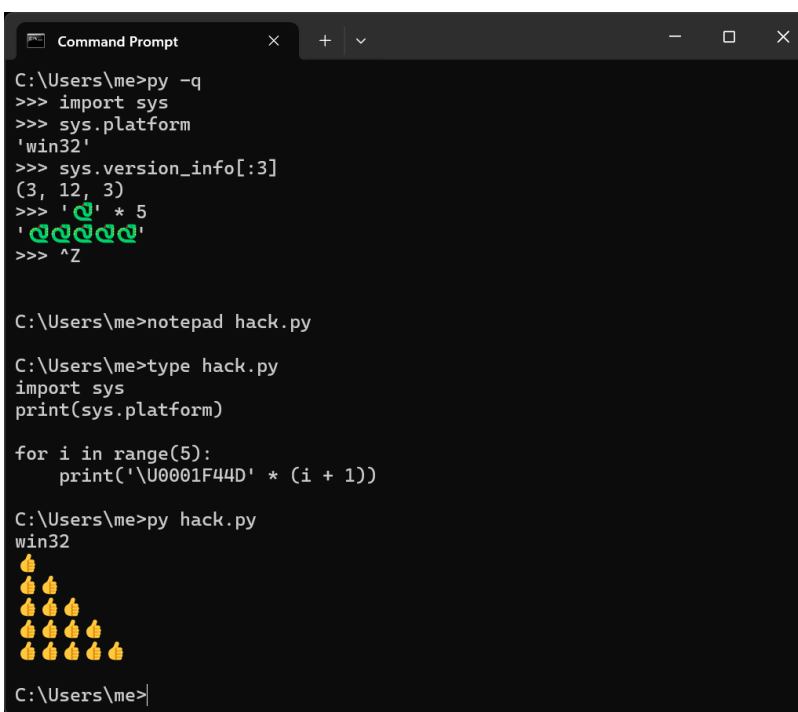


Figure A-1. The *python.org* installer on Windows

Once you’ve installed Python on Windows, *running* it there can be as simple as typing code at a command line and clicking file icons or as complex as learning the nuances of a full-featured IDE. Of these, command lines generally add the least number of moving parts.

To run Python from a command line on Windows, open either *Command Prompt* or *PowerShell* (the non-ISE flavor, normally). Once open, simply type `py` or other options presented in a moment and add a filename to run a file of code (i.e., a *script*).

For example, open Command Prompt from your Start menu (search for it there if needed). Then, type `py` and press Enter to start a Python interactive session where you can type and run code at the `>>>` prompt. To launch a script instead, type `py script.py`, with the name of your script. Figure A-2 demos these commands live on Windows. For space, some demos in this appendix, including this one, use Python’s `-q` flag to suppress messages on session startup; this is cosmetic and optional.



```
Command Prompt
C:\Users\me>py -q
>>> import sys
>>> sys.platform
'win32'
>>> sys.version_info[:3]
(3, 12, 3)
>>> '❤' * 5
'❤❤❤❤❤'
>>> ^Z

C:\Users\me>notepad hack.py

C:\Users\me>type hack.py
import sys
print(sys.platform)

for i in range(5):
    print('\U0001F44D' * (i + 1))

C:\Users\me>py hack.py
win32
❤
❤❤
❤❤❤
❤❤❤❤
❤❤❤❤❤
C:\Users\me>
```

Figure A-2. Running Python by command lines on Windows

The `py` command is technically part of the Python *Windows launcher* that's installed along with Python itself. By default, the launcher runs the most recent Python version installed on your PC, but you can also specify a version to run if there's more than one (e.g., `py -3` runs the latest 3.X, and `py -3.8` runs an older version of it). If you have just one Python or want to use the latest, `py` suffices to launch an interactive session or script.

You can also start Python with command `python` if you opted to add Python to your system PATH during the install, though it's pointless extra typing (`python3` works too, but only if you installed from the Store per the earlier note). And just as on Unix, you can easily save a script's printed output to a file by adding `> filename.txt` to the end of a command (see [Chapter 3](#) for more on such stream redirections).

However, if you opt to go the command-line route, you'll also need to choose a *text editor* to create files of code you wish to save (i.e., scripts to run and modules to import). As demoed in [Figure A-2](#), Windows *Notepad* suffices, but any Windows text editor will fit the bill. To use Notepad, launch it from your Start menu (search there if needed) or by typing its name in a command line, with or without a filename to edit.

Besides command lines, you can also start Python's interactive session by clicking the *Python 3.12 (64-bit)* (or similar) item in its *Start-menu* entry, shown in [Figure A-3](#). This starts the usual Python *REPL* (Read-Eval-Print Loop) interactive session with its `>>>` prompt, just like an explicit `py` command in Command Prompt or PowerShell. You'll still use `py` commands or other techniques, though, to run code files. In all console REPLs on Windows, type or tap the two-key combo `Ctrl+Z` (followed by Enter) at the `>>>` prompt to exit a Python interactive session or simply close the hosting window.

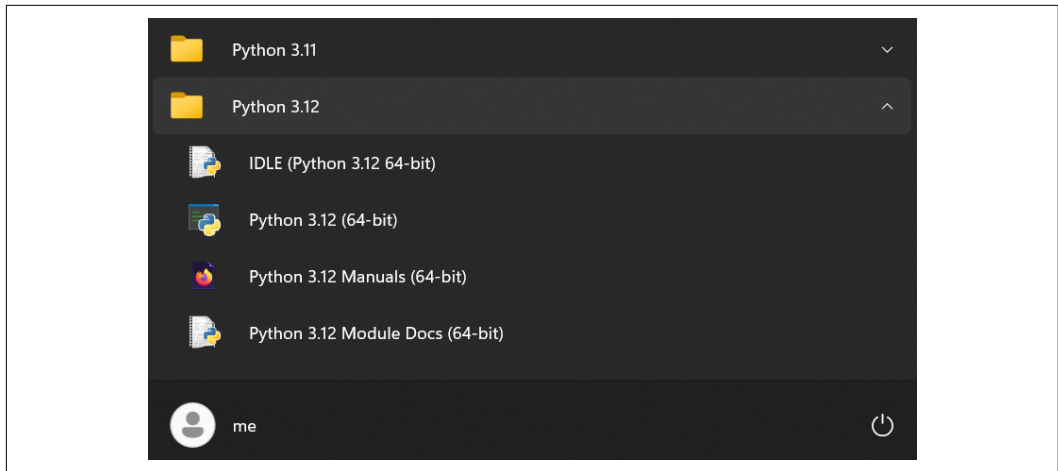


Figure A-3. Python Start-menu options on Windows

If command lines make you break out in hives, you can also run Python from graphical IDEs like PyCharm or Python’s own IDLE. Of these, *IDLE* is included with Python for Windows and provides a simple but sufficient IDE for running this book’s examples. Its utility partly overlaps with command lines (it’s ultimately just a place to type and run code), but it also includes a Python-friendly text editor for code files and simplifies some common coding chores. Notably, it’s able to launch program files without command lines.

As examples, Figures A-4 and A-5 capture IDLE’s interactive “Shell” and editor windows, respectively, with default configurations. You can start IDLE from Python’s entry in your *Start* menu on Windows (try a search for “idle” there to locate and open IDLE quickly). The command `py -m idlelib.idle` also starts IDLE, for reasons covered elsewhere in this book (tl;dr: this is like a module import, but runs instead of importing), and right-clicks on code files in File Explorer can open IDLE too, but require registry edits today.

IDLE’s own Help menu comes with ample usage info that we’ll defer to here, but one tip is worth a callout: a menu *Run*→*Run Module* (or its equivalent F5 shortcut key) in any editor window like that in Figure A-5 lets you launch a script without typing a command line. This runs the code in that window after it’s been saved to a file if needed and routes the code’s printed output back to the Shell window. Its *Run...Customized* version also lets you provide command-line arguments (see `sys.argv` in Python’s manuals for details).

Useful tricks to be sure, but even if you don’t use IDLE to run code this way, it has additional tools we’ll skip here for space, and its code editor alone makes for a compelling alternative to Notepad if you have no other option in mind for scripts and modules.

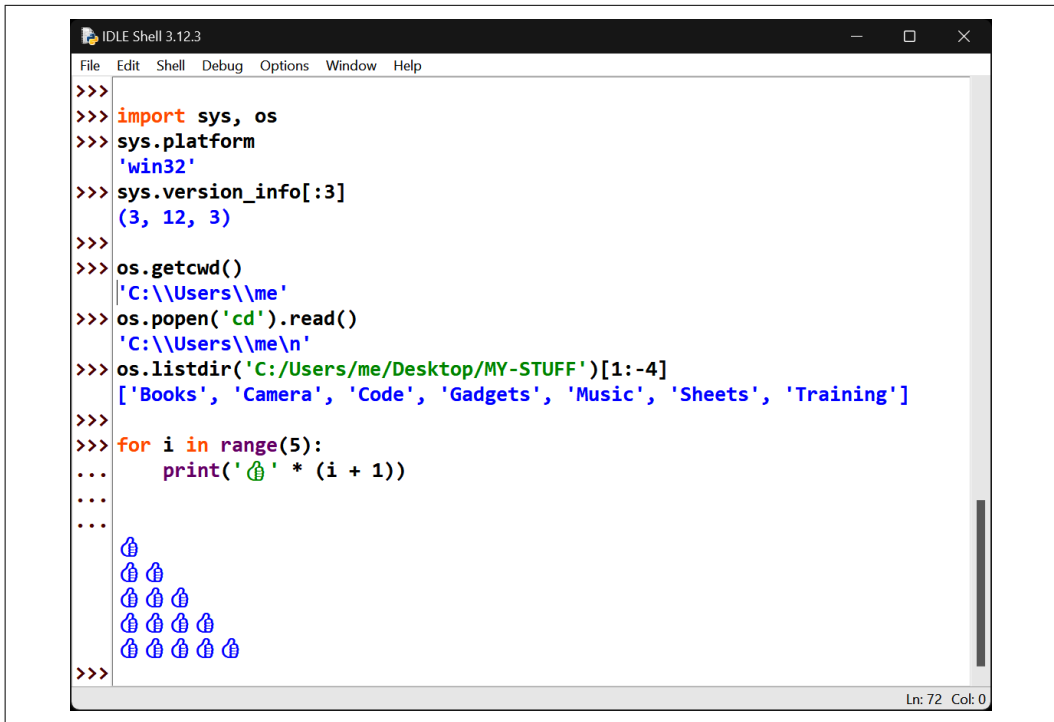


Figure A-4. The IDLE GUI's Shell window on Windows

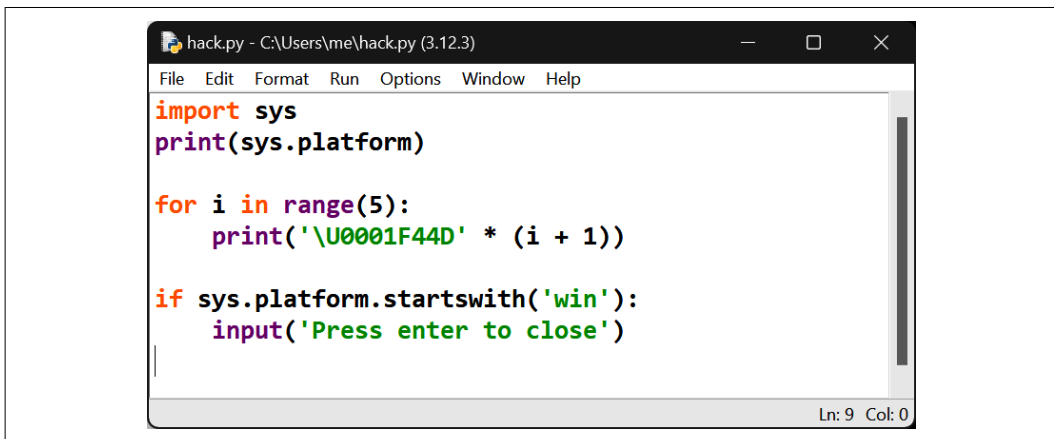


Figure A-5. An IDLE GUI editor window on Windows

Beyond `py`, `python`, `python3`, `Start`, and `IDLE` (which already qualifies as a Windows embarrassment of riches!), a Python program file can also be launched on Windows by typing just its *name* in a command line (e.g., `hack.py`), and by locating and *clicking* its name or icon in Windows File Explorer. These both work thanks to the magic of filename associations in Windows, which Python sets up automatically during its install: any file whose name ends in `.py` is routed to the `py` launcher when named or clicked.

Clicking, however, comes with a drawback: printed *output* of programs you launch this way is lost on program exit, because the program's run window is closed. To keep the window (and hence output) open, simply add a call to Python's `input()` at the bottom of your script to pause for a user Enter-key press. As in [Figure A-5](#), this call can be conditional on the platform to pause selectively (some code may warrant standard-stream TTY tests too).

The `input()` trick won't help if the program commits an *error* (alas, the error messages may perish with the window before the pause is ever reached!), so running by clicks is usually best used only for graphical programs and others that log their errors to files. Tip for GUIs: a `.py` opens a console for standard stream IO when clicked, but a `.pyw` does not.

Before we move on, here are some advanced but useful tips for Python on Windows:

Script #! lines

The `py` Windows launcher treats the *first line* in a script's file as special if it begins with `#!`. This line can name the version and location of the Python to be used to run the file's code (e.g., `#!python3.12`), and all the usual Unix-style lines work (e.g., `#!/usr/bin/python3.12`). This line is entirely optional and no different than naming a Python in the command line used to launch it with `py` (e.g., `py -3.12 hack.py`) but may be useful when there are multiple Pythons installed on your PC, especially when scripts are run with clicks instead of command lines.

Environment variables

Windows command-line interfaces use the `PATH` (a.k.a. `Path`) environment variable to locate named programs like `python`: every folder on this list is searched. This is normally set up for Python automatically during its install if you opt in, but you can tweak it later yourself in Settings (search for “environment variable” there). In the same way, you may also create or modify `PYTHONPATH`, used to locate imported modules (per [Chapter 22](#)), as well as `PYTHONUTF8` and `PYTHONIOENCODING`, used on Windows to specify the default Unicode encoding of files and redirected streams (this convoluted story has changed in 3.X often and will again; see [Chapter 37](#)).

Other Windows options

Most Python code works the same on Windows as on other platforms, especially if it uses Python's portable system tools in modules like `os`. In cases where Windows-specific tools are required, though, the `pywin32` third-party extension allows your Python programs to access many Windows APIs directly.

For more about using Python on Windows, try [python.org's HOWTO](#) as well as the copious resources on the web (with the usual caution about vetting their copious sources). Here, let's move on to the next PC platform on our tour.

Using Python on macOS

As a Unix-based platform, macOS is well suited to Python, open source, and software development in general. At this writing, newer macOS PCs no longer come with a Python preinstalled (and if it seems they do, it's just a stub for installing unrelated toolsets, per the note ahead). Older macOS systems do have a Python, but it's the now-dated 2.X, and may issue a deprecation warning when launched (in other words, you can't use it to run this book's code, and it's not long for the macOS world). Hence, an install is required.

As for Windows, the recommended way to *install* Python 3.X for macOS begins with a visit to the Downloads page at [python.org](#). There, you'll fetch a self-installer for macOS that you'll run to install Python, along with its IDLE GUI and standard library (including the library's `tkinter` GUI toolkit).

The Python you'll get today is a Universal 2 binary that runs natively on macOS PCs using both newer Apple Silicon (ARM) and older Intel (x86) chips, and can be run in the Rosetta 2 emulator (see the web for more on all such terms). **Figure A-6** captures the install process on macOS.



Installation convolution: If you type `python3` in macOS's Terminal *before* running the *python.org* install, you may get an Apple popup that asks if you want to install Python as part of Xcode's "command line developer tools." This is similar in spirit to the Microsoft Store redirect on Windows of the preceding section—and similarly confusing!

On macOS, though, most users should ignore the offer and instead install Python from *python.org* as described here because it makes your Python independent of the version and configuration choices made by a tools package. This is also true if you already have Xcode and its Python: install a new Python from *python.org* for generally better control.

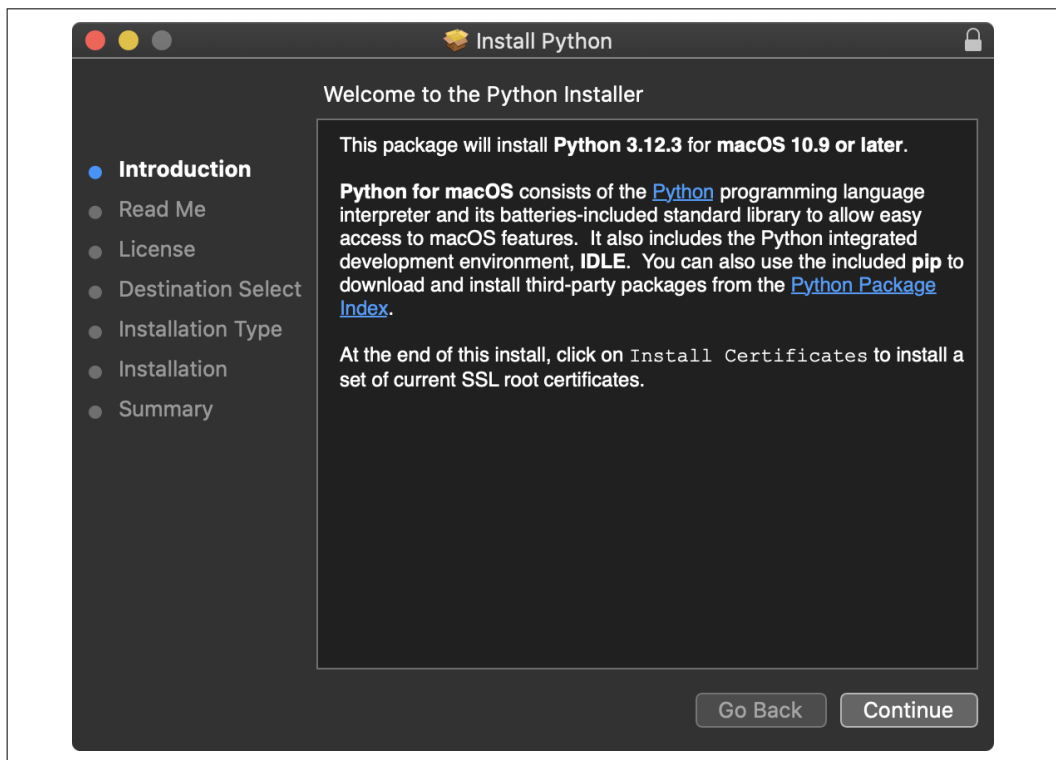
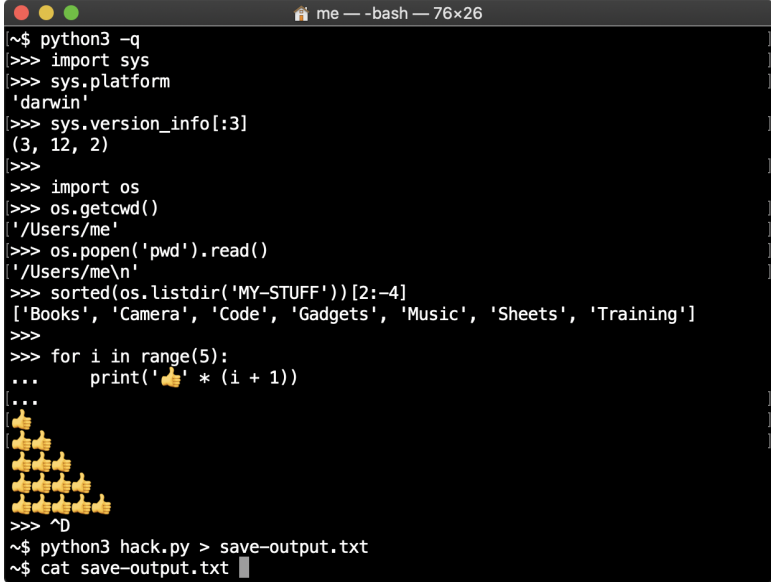


Figure A-6. The *python.org* installer on macOS

After the install, you can *start* Python both with its entries in *Launchpad*, which mostly mirrors your PC's *Applications* folder in *Finder*, as well as command lines in *Terminal*, which provides a standard Unix command-line shell. Of these, *Terminal* may be the most basic way to get started with this book's examples on macOS.

To run code by command line on macOS, open Terminal by clicking its entry in either Launchpad's *Other* folder, or *Applications*→*Utilities* in Finder. Then, type `python3` to start a Python interactive session where you can type and run code, or `python3 script.py` to launch a code file. This works because `python3` is added to your system PATH by the install (and as a caution, `python` may mean Python 2.X on older PCs).

Figure A-7 demos Python commands live on macOS. As usual on Unix-based systems, type keys combo Control+D at the `>>>` prompt to exit a Python interactive session here (yes, this differs from Windows), and alias `python3` to something shorter in your shell's startup files if seven characters is too much (e.g., alias to `py`, if you want to avoid some disorientation when hopping to and from Windows).



```
~$ python3 -q
>>> import sys
>>> sys.platform
'darwin'
>>> sys.version_info[:3]
(3, 12, 2)
>>>
>>> import os
>>> os.getcwd()
'/Users/me'
>>> os.popen('pwd').read()
'/Users/me\n'
>>> sorted(os.listdir('MY-STUFF'))[2:-4]
['Books', 'Camera', 'Code', 'Gadgets', 'Music', 'Sheets', 'Training']
>>>
>>> for i in range(5):
...     print('👍' * (i + 1))
...
👍
👍👍
👍👍👍
👍👍👍👍
👍👍👍👍👍
>>> ^D
~$ python3 hack.py > save-output.txt
~$ cat save-output.txt
```

Figure A-7. Running Python in macOS Terminal

When using command lines to run code, you'll also use a *text editor* to create files of Python code (scripts and modules) on macOS. Its built-in TextEdit suffices but isn't very code-friendly out of the box (e.g., you'll want to set a monospace font right away). Any macOS text editor is up to the task of editing Python code, including *IDLE* (up next), the *vi* and *nano* command-line-based editors familiar to Unix users, and other options you can explore on the web.

After the install on macOS, you'll also find two tools for running Python code in other modes, available in both *Launchpad* and your *Applications* folder in *Finder*. As captured in both Figures A-8 and A-9, the *Python Launcher* allows you to run a file of Python code with either a *click* in Finder or a *drag* to its icon or name, and *IDLE* provides a basic edit-and-run IDE GUI for Python code. (Python itself, invoked by a `python3` command, shows up in `/Library/Frameworks`, though you don't normally need to care.)

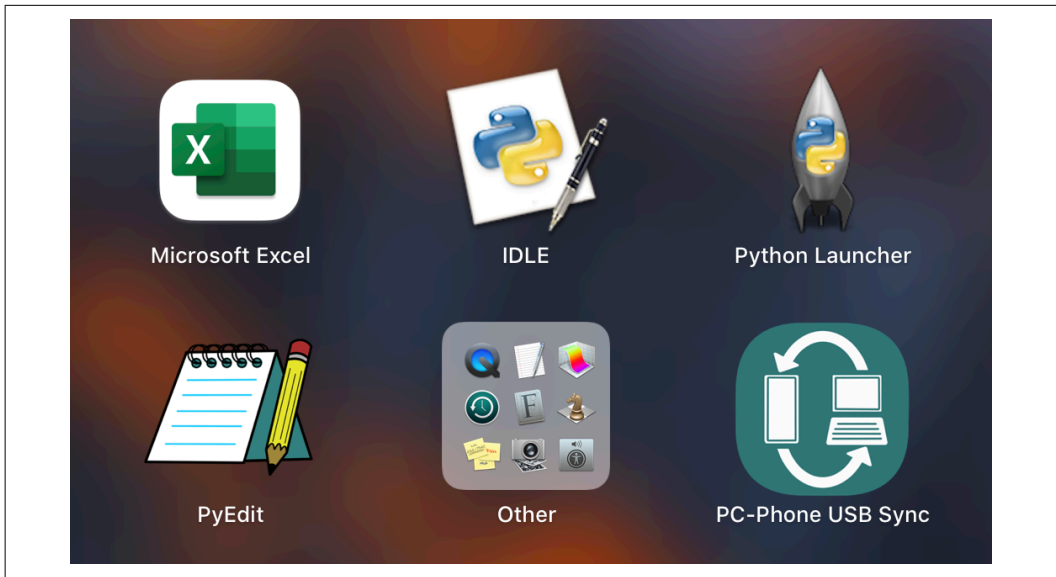


Figure A-8. Python's IDLE and launcher in macOS Launchpad

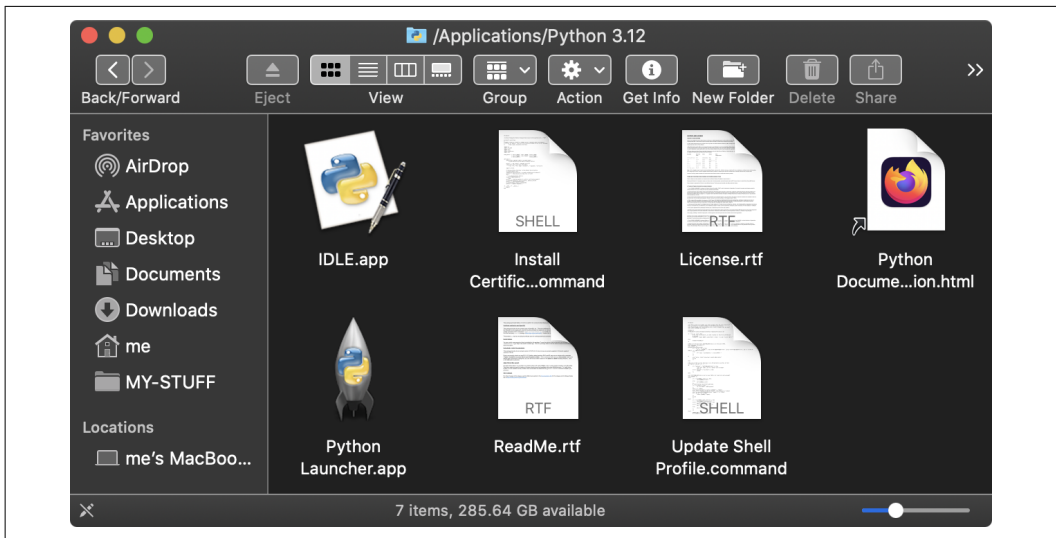
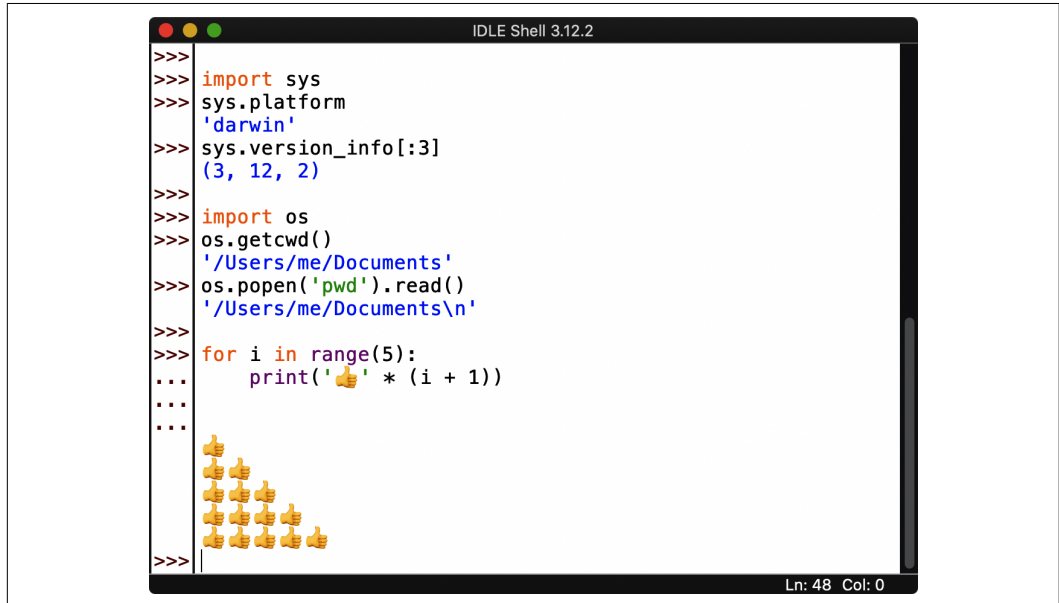


Figure A-9. Python's Applications folder in macOS Finder

Figure A-10 shows IDLE on macOS. Launch it most easily with its Launchpad or Finder entries or by right-clicking (a.k.a. control-clicking) any Python code file in Finder and choosing IDLE in Open With (you can make this association permanent there if desired, and global in Finder's Get Info).

Because IDLE is coded in Python as a tkinter GUI, it looks and works the same on all supported platforms (essentially, all PCs sans mobile heroics). As elsewhere, it allows you to edit and run files of Python code without having to use command lines or other editors. See IDLE's earlier Windows

coverage for more tips; as noted there, IDLE can serve as a Python-friendly code editor, even if you run files by command lines, clicks, or drags.

The image shows a screenshot of the IDLE Shell window on macOS. The window title is "IDLE Shell 3.12.2". The code being executed is as follows:

```
>>> import sys
>>> sys.platform
'darwin'
>>> sys.version_info[:3]
(3, 12, 2)
>>> import os
>>> os.getcwd()
'/Users/me/Documents'
>>> os.popen('pwd').read()
'/Users/me/Documents\n'
>>> for i in range(5):
...     print('👍' * (i + 1))
...
...
>>>
```

The output of the code is a series of thumbs-up emojis arranged in a triangular pattern:

```
👍
👍👍
👍👍👍
👍👍👍👍
👍👍👍👍👍
```

The status bar at the bottom right of the window shows "Ln: 48 Col: 0".

Figure A-10. The IDLE GUI's Shell window on macOS

To wrap up, here are a handful of advanced usage tips, with macOS spins that also apply to other Unix platforms like Linux and Android coming up next:

Script `#!` lines

If the *first line* of a script begins with `#!`, it's treated as special on macOS, just as it is on Windows. On macOS, this is a function of the shell (e.g., *Bash* or *Zsh*) that's running your command lines, not Python. For instance, a top-of-script line `#!/usr/bin/python3.12` tells the shell which Python should run the rest of the file's lines. This isn't different than naming your Python in a command line explicitly (e.g., `/usr/bin/python3.12 hack.py`), but a `#!` line allows a script to be run by just its name (e.g., `hack.py`) if it's also made executable (see `chmod`).

Running by clicks

Python code files run with a *click* in Finder, but you may have to choose the Python Launcher in Open With and make it permanent with Always (or Get Info). Unlike on Windows, output is retained in the resulting console window on exit and errors. A `#!` first line isn't required but can be used.

Environment variables

On macOS, you can also set or change environment variables like `PATH`—used to locate programs like `python3` named in command lines, and `PYTHONPATH`—used to locate imported modules—with code in your shell's startup files (e.g. `~/.bash_profile` or `~/.zprofile`). `PATH` is set automatically by *python.org* installs. For pointers on the shell code used to set these variables, see the example in [Chapter 22](#), as well as the web and your PC's docs (e.g., `info bash` and the unfortunately coded `man bash`).

Other macOS options

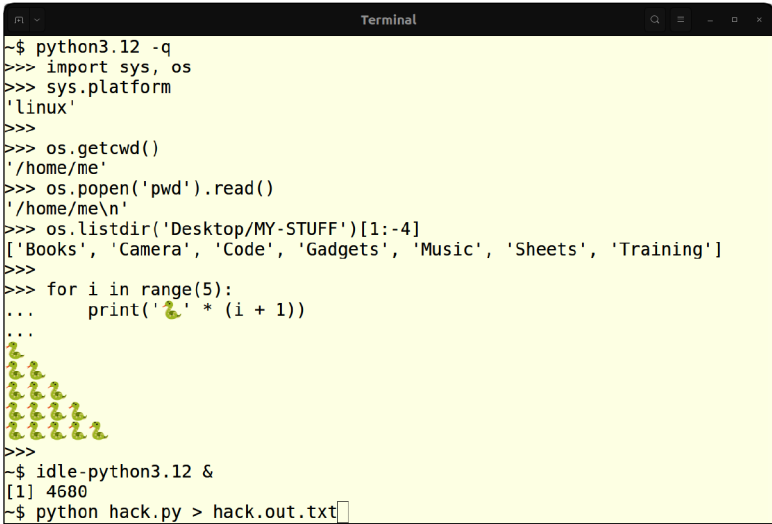
For completeness, it's worth noting that there are additional platform-specific tools for Python on macOS (e.g., *PyObjC*), and the third-party *Homebrew* package manager provides an entirely different install scheme for Python on macOS, which works equally well but has extra setup steps that make it better suited to advanced readers.

For space, we'll skip further details here; see the [HOWTO](#) at *python.org* and your local web search engine for more Python options on macOS.

Using Python on Linux

Python is a staple on Linux, where it's used both for user applications and system tools. Because it's so ubiquitous on this Unix-based platform, Python may come preinstalled with your Linux distribution; type `python3` in a shell window (e.g., *Terminal*) to check. If you need to install manually, do the usual thing for your Linux flavor: a `sudo apt install python3` in Terminal does the deed in Ubuntu distributions (try `yum` on some others).

Once you've got a Python 3.X installed, run code interactively and launch code files with the usual Terminal command lines, like those captured in [Figure A-11](#). A `python3` (or a version-specific name) starts an interactive session for typing and running code, and adding a filename (e.g., `python3 script.py`) runs a file. As on other platforms, your `PATH` setting is used by such commands to locate Python. As on all Unixes, the key combo `Ctrl+D` at `>>>` exits a Python REPL, and shorter shell aliases for `python3` can avoid some typing.



```
Terminal
~$ python3.12 -q
>>> import sys, os
>>> sys.platform
'linux'
>>>
>>> os.getcwd()
'/home/me'
>>> os.popen('pwd').read()
'/home/me\n'
>>> os.listdir('Desktop/MY-STUFF')[1:-4]
['Books', 'Camera', 'Code', 'Gadgets', 'Music', 'Sheets', 'Training']
>>>
>>> for i in range(5):
...     print('🐱' * (i + 1))
...
🐱
🐱🐱
🐱🐱🐱
🐱🐱🐱🐱
🐱🐱🐱🐱🐱
>>>
~$ idle-python3.12 &
[1] 4680
~$ python hack.py > hack.out.txt
```

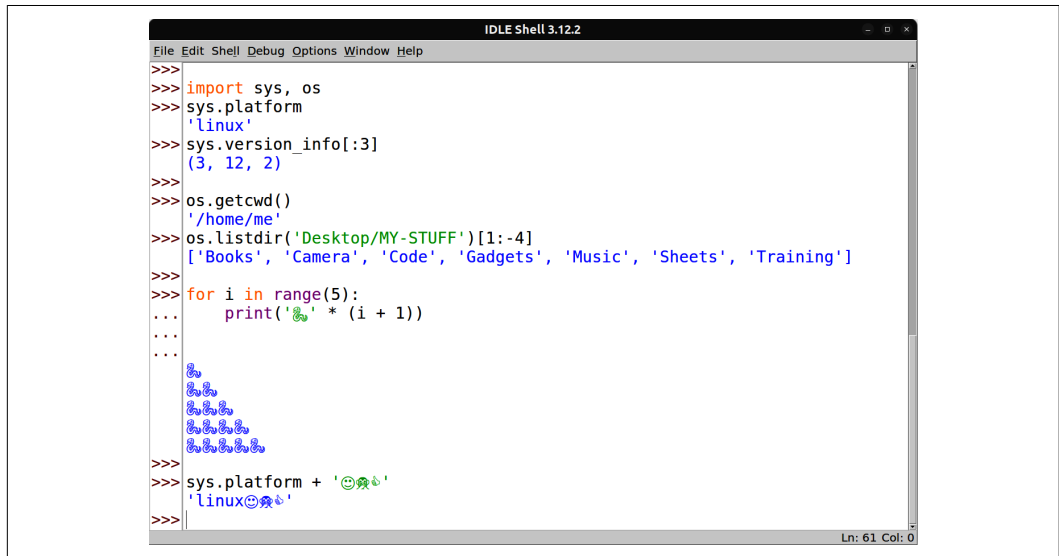
Figure A-11. Running Python in Linux Terminal

When using command lines, you'll also need to use a text editor for scripts and modules. To make such a file of Python code, any Linux text editor will do, including the graphical *Gedit* default on Ubuntu and the shell-oriented *vi* and *nano*.

You can also use Python's *IDLE* edit-and-run GUI on Linux. Launch it with an `idle` command line after installing it with `sudo apt install idle3`, or similar on other distributions. A code-file right-click and Open With in file explorers may start IDLE too.

IDLE fine print: you may need to force versions with a more specific name (e.g., `idle-python3.12`); emojis might not work until a font install (e.g., `sudo apt install ttf-ancient-fonts-symbola`); and a platform-agnostic command line `python3 -m idlelib.idle` starts IDLE, too, per the Windows flavor noted earlier.

Figure A-12 demos IDLE running on an Ubuntu Linux PC after all the kinks have been ironed out; it works the same on Linux as on Windows and macOS, and is covered in more detail in this appendix's Windows section.



```
File Edit Shell Debug Options Window Help
>>>
>>> import sys, os
>>> sys.platform
'linux'
>>> sys.version_info[:3]
(3, 12, 2)
>>>
>>> os.getcwd()
'/home/me'
>>> os.listdir('Desktop/MY-STUFF')[1:-4]
['Books', 'Camera', 'Code', 'Gadgets', 'Music', 'Sheets', 'Training']
>>>
>>> for i in range(5):
...     print('🐍' * (i + 1))
...
...
🐍
🐍🐍
🐍🐍🐍
🐍🐍🐍🐍
🐍🐍🐍🐍🐍
>>>
>>> sys.platform + '🐍🐍'
'linux🐍🐍'
>>>
```

Figure A-12. The IDLE GUI's Shell window on Linux

Also noteworthy on Linux:

- If you wish to use Python's portable `tkinter` GUI toolkit, you can install it separately if needed with `sudo apt install python3-tk` (or similar, in the richly bifurcated world of Linux package installs).
- As on macOS, a line starting with `#!` at the top of your script can denote which Python runs the file, and environment variables like `PATH` and `PYTHONPATH` can be set in shell startup files, but the former is not usually required. See the macOS section's coverage of both topics and [Chapter 22's](#) `PYTHONPATH` example.
- Python files can be run by *clicks* on Linux too, but details vary. In Ubuntu's Files, “Run as a Program” runs a clicked Python file that has both executable permission (e.g., `chmod +x script.py`), and a `#!/...` first line that gives the path to Python, but the Windows caution about output disappearing on exit or error applies.

- It's not uncommon on Linux to build Python from its source code distribution, available at either python.org or GitHub. This entails a few simple command lines (`configure` and `make`) but is beyond the scope of both this chapter and most Python beginners; see the Downloads page at python.org for code and details.

For more info about Python on Linux, try the web at large or python.org's [HOWTO](#). Here, it's time to move ahead to Python's story on mobile.

Using Python on Android

Android is a secure derivative of Linux adapted for the unique constraints of mobile devices, and it is the most widely used operating system in the world at this writing. Despite this platform's Java and Kotlin programming-language biases, Python can be used as a first-class programming citizen on Android devices in both learning and development roles. This book's examples, for instance, will work well on your Android phone or tablet.

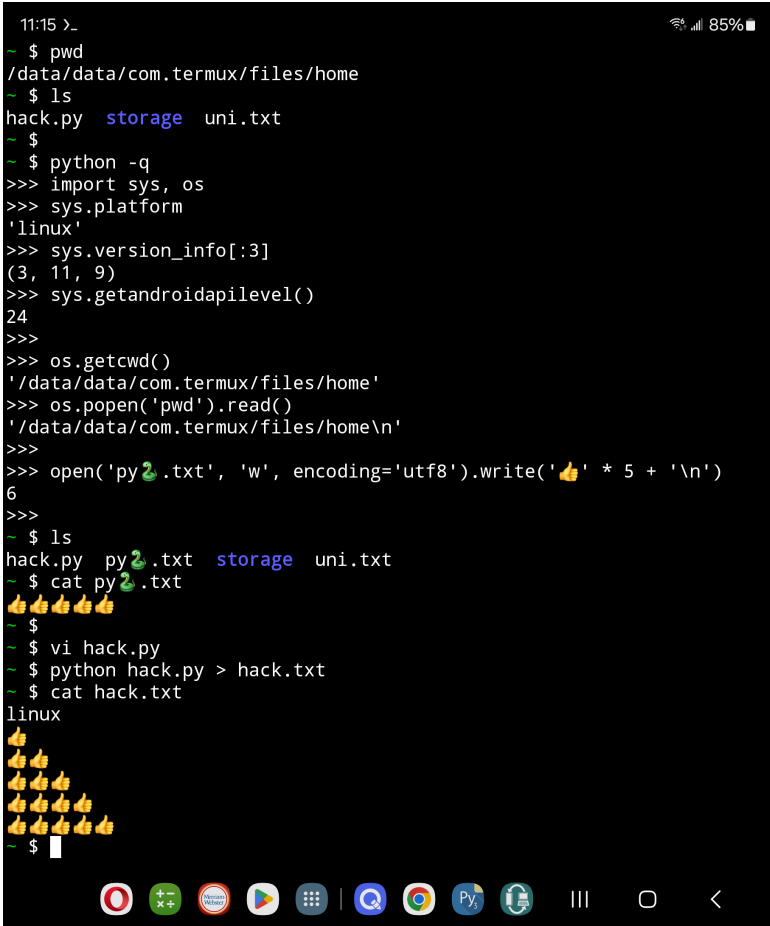
To run Python locally on your Android, you'll first install an app that supports it from an app store like *Play* or *F-Droid*. Among these apps, *Termux*, *Pydroid 3*, and *QPython* all allow you to run Python code on Android directly in multiple modes. While we can't do justice to these and other Python apps, here's a quick rundown of two to get you started.

The free and open source *Termux* app for Android provides a full-featured Linux shell, toolset, and package manager. To use Python in Termux, first install the Termux app from the **F-Droid** store (its Play version is defunct). Then, open the app from your Apps screen, and install Python 3.X inside it with a `pkg install python` command line in its shell.

Termux opens with a standard *Bash* command-line shell, where you can tap out commands to launch an interactive Python session with `python`, and run a file of code by adding a filename (e.g., `python script.py`). Stream redirection works as on all Unix, and command `python3` is the same as `python` if you prefer Unix uniformity and don't mind the extra tap. Both commands are automatically usable post install without `PATH` mods.

You can use *code files* (scripts and modules) located in any folder Termux has access to (which generally means shared or app-private storage, per ahead) and make and change them either with separate text editor apps or within Termux itself using Linux text editors like *vi* and *nano* (install them in Termux with `pkg install` as needed). Termux also supports Android's Storage Access Framework to make its app-private storage visible to some file-explorer apps, although shared storage is more accessible and usable.

Figure A-13 demos the Termux app running Python code and file on Android. As on all Unixes, the keys combo `Ctrl+D` at `>>>` ends a Python interactive session in Termux (via Termux's *CTRL* button or keyboards ahead), as does killing the app, and shell aliases can shorten the `python` (or `python3`) command. With apps, you'll generally use the version of Python provided; in Termux, this means one of the versions in package repos, but you may be able to build a newer one from source code. You can also install a host of extensions to use in your code by command line, with both Termux's `pkg` and Python's `pip`.



```
11:15 >_
~ $ pwd
/data/data/com.termux/files/home
~ $ ls
hack.py  storage  uni.txt
~ $
~ $ python -q
>>> import sys, os
>>> sys.platform
'linux'
>>> sys.version_info[:3]
(3, 11, 9)
>>> sys.getandroidapilevel()
24
>>>
>>> os.getcwd()
'/data/data/com.termux/files/home'
>>> os.popen('pwd').read()
'/data/data/com.termux/files/home\n'
>>>
>>> open('py👍.txt', 'w', encoding='utf8').write('👍' * 5 + '\n')
6
>>>
~ $ ls
hack.py  py👍.txt  storage  uni.txt
~ $ cat py👍.txt
👍👍👍👍👍
~ $
~ $ vi hack.py
~ $ python hack.py > hack.txt
~ $ cat hack.txt
linux
👍
👍👍
👍👍👍
👍👍👍👍
👍👍👍👍👍
~ $
```

Figure A-13. Running Python in the Termux app on Android

Termux may be the path of least resistance for getting started with Python on Android. It has more features we'll largely skip here, including home-screen widgets that run Python scripts on taps, and all Linux concepts covered earlier apply, including PYTHONPATH and PATH environment-variable settings. Its chief downside for some users may be that it is limited to command lines sans its optional X Window System support, which is considerably complex to use; *IDLE*, for example, would be difficult at best to run in Termux.

If you're looking for something a bit more graphical, the *Pydroid 3* app also provides a command-line shell and interactive Python session, but it adds a GUI IDE for editing and launching Python code. The IDE's edit/run window is captured in [Figure A-14](#).

Pydroid 3 is today installed from [Play](#). Its shell and interactive session are less user-friendly than the richer command-line support in Termux, but its IDE may seem more comfortable for users unaccustomed to command lines. Similar in spirit to *IDLE* on PCs, this app's IDE allows you to edit Python code, and launch it with a simple (and yellow) button press.

```

11:16 11:16 85%
frigcal.py
/storage/emulated/0/Download/Frigcal-source/frigcal.py

287 if TkVersion <= 8.6:
288     text = ''.join((ch if ord(ch) <= 0xFFFF else '\uFFFD') for ch in text)
289     return text
290
291
292 #=====
293 # Month display window: main and clones
294 #=====
295
296 class MonthWindow:
297     """
298     the main display, with its state and callback handlers:
299     - created by main() and Clone button, kept on OpenMonthWindows;
300     - uses local ViewDateManager object to manage viewed date and day
301     - creates local EventDialog subclass dialogs on user actions and
302     - uses CalendarsTable and EventsTable globals, created by ics file
303     - subclassed to customize onQuit for popup Clone windows to close
304     """
305
306     def __init__(self, root, startdate=None, windowtype='Main'):
307
308         # window's state informaton
309         self.root = root # the Tk (or a Toplevel) main window
310         self.monthlabel = None # monthname label, for refills
311         self.daywidgets = [] # [(dayframe, daynumlabel)], a
312         self.eventwidgets = {} # {uid: evententry}, all disp
313         self.tandemvar = None # if get(), all windows respon
314
315         # set up current view date data
316         self.viewdate = ViewDateManager() # displayed month date a
317         self.viewdate.settoday() # initialize date object
318         if startdate:
319             self.viewdate.setdate('%s/%s/%4s' % startdate.mdy())
320
321         # more options state information

```

Figure A-14. The Pydroid 3 app's IDE on Android

On top of its IDE, Pydroid 3 adds support for many popular tools, including scientific-programming libraries and, remarkably, Python's tkinter GUI toolkit. As in Termux, Python's version is preset in Pydroid 3 (though source builds are elusive), and Python's pip is available to install extensions (though with a dedicated GUI in this app).

Fair warning: as a substantial trade-off, Pydroid 3 is also a *freemium* app, which will flash rude full-page ads at you unless and until you pay a required fee—an unfortunately common paradigm in Android, which you'll have to weigh for yourself. In addition, Pydroid 3 has a history of waffling on support for *storage* access in response to Android and Play edicts. By contrast, *Termux* today is entirely free and void of ads, supports broad storage access, and chooses alternative app stores rather than limiting functionality for Android changes mandated by Play.

See the web and app stores for info about other Python programming apps on Android omitted here for space. While you're at a store, you may also want to explore text editor apps like *QuickEdit*—which is able to colorize and run Python code; and alternative onscreen keyboards like *Hacker's Keyboard*—which adds PC keys not available in stock options but commonly used for coding (e.g., arrows and Ctrl). Some Python apps also include tools to augment onscreen keyboards that can be tailored or disabled, and Bluetooth keyboards and casting to larger screens can naturally aid usability too.

It's also worth noting in closing that *CPython* plans to add Android to its list of officially supported platforms soon, which may foster additional options going forward. Moreover, although most beginners will use an app to run Python code on Android as described, it's also possible to build standalone apps for Android that are coded in Python but used like any other app. We'll return to this option at the end of this appendix after one last platform.

Android's Proprietary World

Python programmers should also be aware that Android imposes numerous constraints on apps, some of which may seem onerous to developers with backgrounds in more interoperable platforms. Most of these constraints are rationalized on the grounds of security or performance, but all reduce utility.

Android's *storage*, for instance, is split into a shared and persistent area with controlled access, along with areas partly or wholly private to apps that may vaporize on app uninstalls. Hence, while POSIX file tools and paths do work on Android, Python code must take care to either use accessible folders or run proprietary Java API tools that request or use enhanced permissions.

In addition, background or long-running *processes* may run afoul of limits; opinionated choices of *tools and languages* are nearly imposed on developers; and *throttling* for power, heat, memory, or other bias is a norm on most phones.

On the upside, Android users can install apps outside its owner's store and can access the filesystem with numerous file-explorer apps. That makes Android more open than iOS today, but this is a large and fluid topic. If you care about using Python on mobiles, be sure to watch other resources for news on this front.

Using Python on iOS

iOS—which includes its iPadOS offshoot in this guide—is a macOS derivative targeted at mobile devices. Like Android, it has limiting biases for programming languages (Swift and Objective-C), and it is even more strict about carving up storage into restricted app sandboxes with proprietary access rules and tools. Despite these constraints, though, your iPhone or iPad can be used to run Python code, too, including the code in this book.

Like Android, you'll normally use Python on your iOS devices by installing an app that runs Python code. Among these, *Pythonista 3* provides an interactive Python session and a GUI for editing and running files of code, as in other IDEs. In addition, this app comes with access to native iOS features and a toolkit for building GUIs in Python for iOS. It also can share code files with the *Files* app to be opened with taps.

To vet for yourself, fetch Pythonista 3 from the [App Store](#). [Figure A-15](#) shows this app in action (yes, on a humble and historical iPod). Be sure to also explore the other iOS options on the store; the *Pyto* app,

for example, provides similar functionality, and comes with the *Toga* UI library for coding portable GUIs (there's more on Toga at standalone apps ahead).

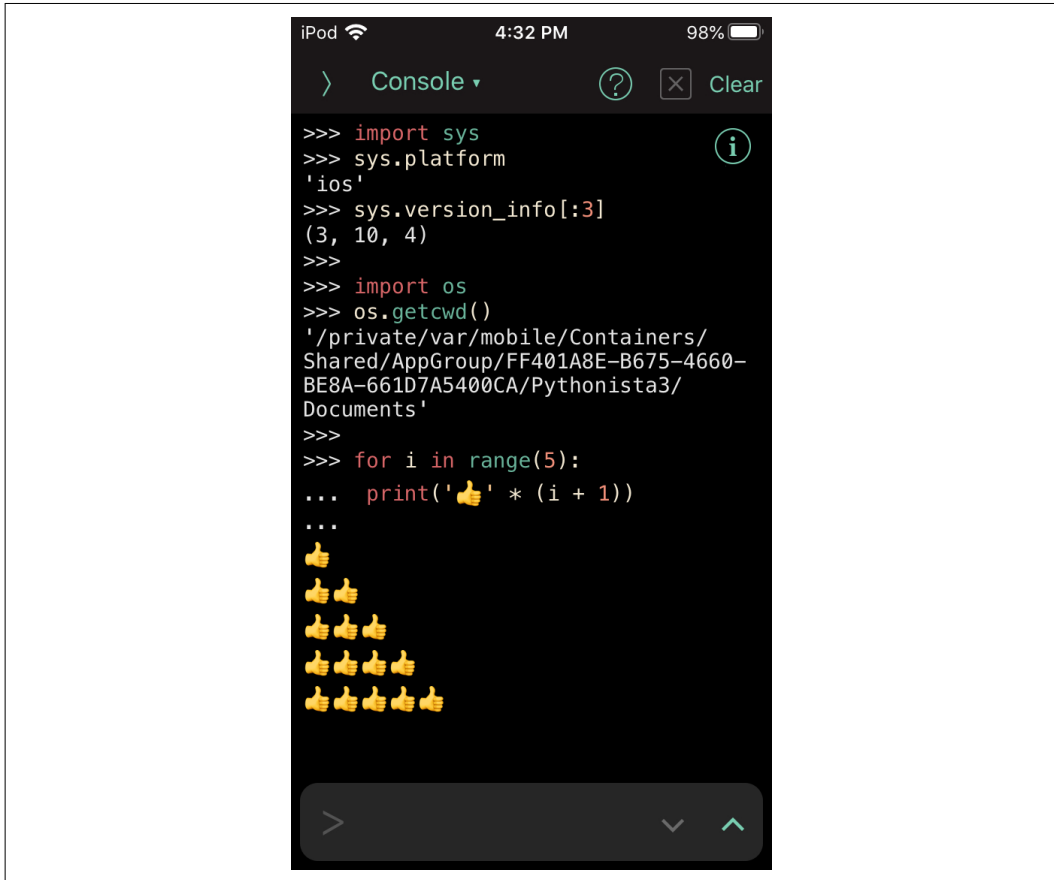


Figure A-15. Running Python in the Pythonista 3 app on iOS

Apart from app choices and platform restrictions, using Python on an iOS device is largely the same as on Android and PCs, so we'll skip further details here. For more on using Python for iPhone and iPad, see the Apple App Store and the web at large.

Like Android, iOS is also scheduled to be granted officially supported status in *CPython* soon, which may yield options impossible to predict today; watch the web for new developments. Also like Android, it's possible to package your Python programs as standalone apps for iOS, but we must move on to this appendix's next section to see how.

Standalone Apps and Executables

Besides running Python source code with the traditional schemes we've just met, it's also possible to bundle Python code into a standalone program that users run the same way they run any other program on their device (e.g., by a click or tap). In fact, users can't even tell these bundles are written in

Python at all: no source code is visible, no other installs or apps are required, and changes in a locally installed Python have no effect on the bundle.

The way you'll build standalones varies per platform. As a noncomprehensive sample of prominent tools today, you can build standalone executables for Windows and Linux with *PyInstaller*; standalone apps for macOS with *PyInstaller* and *py2app*; and standalone apps for Android and iOS with *Buildozer* and *Briefcase* (the latter also offers options for PCs).

On Android, for instance, it's possible to develop standalone apps completely in Python. Although this takes more effort than running code in another app, its products are fully functional and idiomatic GUI apps coded in Python, which run with a tap, leverage Android APIs when needed, and can be both side-loaded and uploaded to app stores.

As a demo, [Figure A-16](#) captures one of many Python-coded standalone apps for Android, running on a foldable. This app, made by this book's author, is freely available in the Play store; is built for Android with *Buildozer*; uses the portable *Kivy* toolkit for its GUI; and relies on Kivy's *pyjnius* to access Android Java APIs when required in a small fraction of its code (e.g., to request permissions, run services, open docs, and get drive labels).

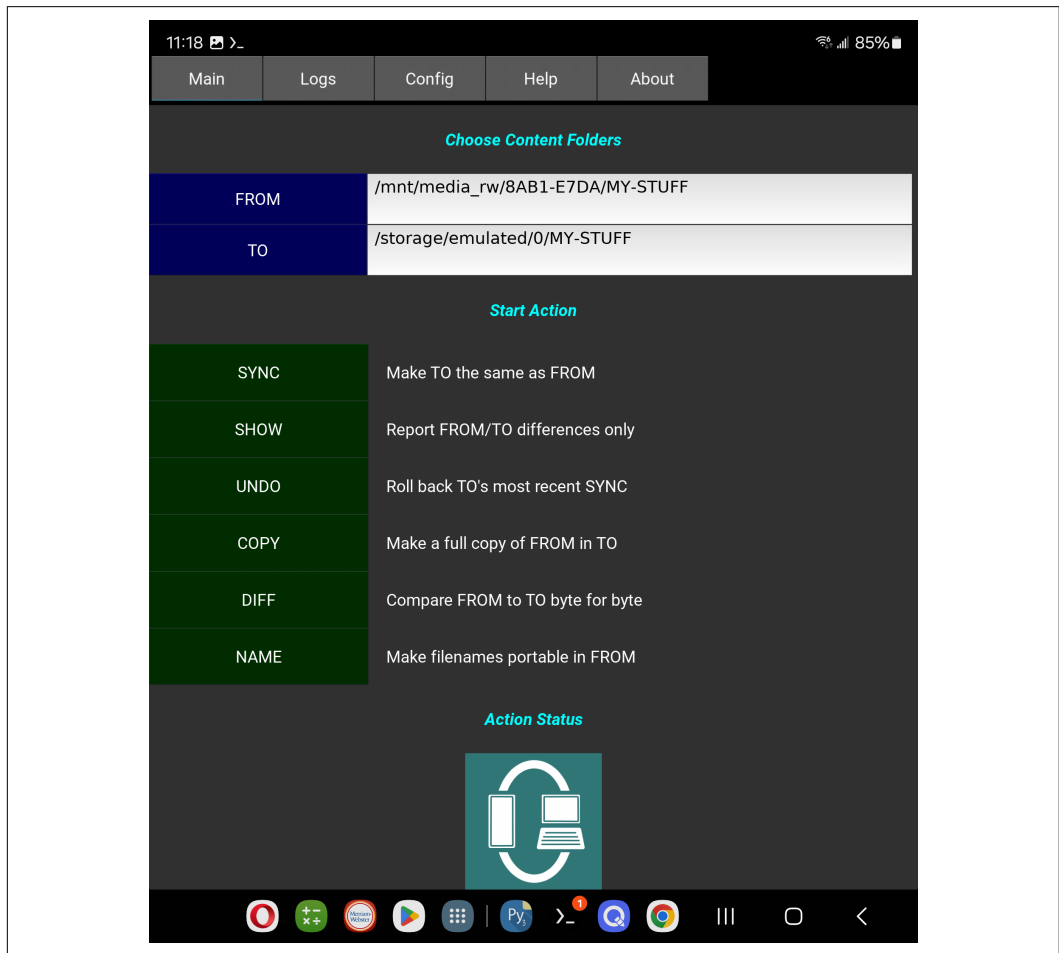


Figure A-16. A Python-coded standalone app running on Android

Crucially, such apps can also run on PC platforms—Windows, macOS, and Linux—from *the same code base*. **Figure A-17**, for instance, shows the same app running on macOS. Its code is bundled for macOS and other PCs with *PyInstaller*; its Kivy GUI is automatically cross-platform; and its POSIX file-sync code works everywhere. The net result is a Python-coded app that runs across a range of PC and mobile hosts, with native behavior on each.

To see this for yourself, fetch this app’s Android version on Play and its PC versions at this book’s website or quixotely.com. Disclaimer: if the Android app is unavailable on Play, check for it at the latter two sites or try a web search; book lifespans tend to be substantially longer than those of apps dependent on stores and platforms (see Termux’s troubles!).

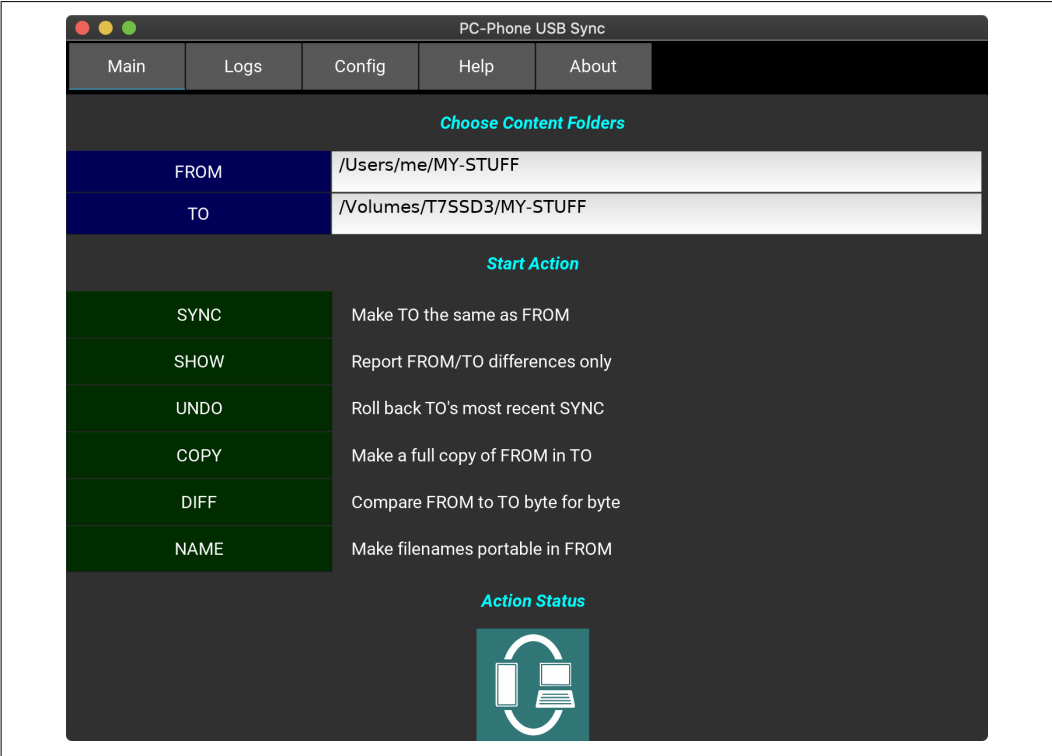


Figure A-17. The same app running on macOS

Nor is this toolset the only interoperability game in town. The alternative *BeeWare*, with its portable *Toga* GUI toolkit and *Briefcase* app builder, promises similar platform independence and advertises additional packaging options on PCs. Moreover, some apps built with such tools can work on iOS, too, though its lack of a user-accessible filesystem renders much cross-platform code unusable (e.g., POSIX file-path syncs are impossible).

The takeaway here: with a portable programming tool like Python, you’re not locked into a single platform’s proprietary realm—unless, that is, you develop for platforms that disqualify code that runs anywhere else. As always, choose your coding battles wisely. Security counts, but closed platforms enable monopolies and stifle innovation.

Standalones may not be very useful when you're just getting started with Python (and make no sense at all for running the examples in this book!), but they may become more important when you start writing programs for others to use. When you're ready to explore standalone deliverables in Python, see the web for current tools and details in this domain.

Etcetera

While the platform techniques we've explored here are perhaps the simplest and most common ways to use Python, there's much more to this story. For instance, this appendix hasn't said anything about using Python in:

- Other IDEs like *PyCharm*, *PyDev*, *Wing*, and *VSCode*
- Web-based notebooks like *IPython* and *Jupyter*
- Alternative Python implementations like *PyPy*, *Cython*, *Numba*, and *Jython*
- Alternative Python distributions like *Anaconda* and *ActiveState*
- The cells and macros of spreadsheets like *Excel*
- Web servers using frameworks like *Flask* and *Django*
- Web browsers using the emerging *WebAssembly* and *Pyodide*

And lots of other options in no way judged by omission here. This book visits some of these in [Chapter 1](#), summarizes Python implementations in [Chapter 2](#), briefly reviews *Jupyter* and *WebAssembly* in [Chapter 3](#), and uses *PyPy* for benchmarks in [Chapter 21](#). In general, though, advanced usage contexts like these are interesting but out of scope for this Python fundamentals text, and best deferred until you've mastered the language itself.

In the end, Python usage details and options tend to evolve as rapidly as Python itself. Indeed, each prior edition of this book has had to revise its usage coverage radically, and this one expects to fare no better. As noted at the start of this appendix, you should expect to check both Python's docs and the web at large for new-and-exciting developments almost certain to emerge by the time you read these words.

Solutions to End-of-Part Exercises

This appendix provides solutions for the book’s end-of-part exercises. Code files named by captions or narrative in these solutions are available in the book examples package’s *AppendixB* folder, which has one subfolder per part (e.g., *AppendixB/Part1* is the first part’s files). See the [Preface](#) for more info on the examples package.

Part I, Getting Started

See “[Test Your Knowledge: Part I Exercises](#)” on page 48 in [Chapter 3](#) for the exercises.

1. *Interaction:* Assuming Python is configured properly, the interaction should look something like the following. You can run this any way you like—in IDLE, a console, an app, a notebook’s page, and so on:

```
$ python3
...information lines...
>>> 'Hello World!'
'Hello World!'
>>>                                     # Use ctrl+D/ctrl+Z to exit on Unix/Windows, or close window
```

2. *Programs:* Your code (i.e., module) file should look something like [Example B-1](#):

Example B-1. Part1/module1.py

```
print('Hello module world!')
```

And here is the sort of interaction you should have; for console launches, be sure to use your platform’s version of the “python3” command (e.g., try “py -3” on Windows):

```
$ python3 module1.py
Hello module world!
```

Again, feel free to run this other ways—by clicking or tapping the file’s icon, by using IDLE’s *Run→Run Module* menu option, by UI options in web notebooks or other IDEs, and so on.

3. *Modules*: The following interaction listing illustrates running a module file by importing it:

```
$ python3
>>> import module1
Hello module world!
>>>
```

Remember that you will need to *reload* the module to run it again without stopping and restarting the interactive interpreter (i.e., REPL). Moving the `.py` file to a different directory and importing it normally fails: Python likely generated a `module1.*.pyc` file in the `__pycache__` subdirectory of the source code file's folder, but it won't use it when you import the module there if the source code (`.py`) file has been moved elsewhere and to a folder not in Python's import search path.

The `.pyc` file is written automatically if Python has access to the source file's directory; it contains the compiled bytecode version of a module. See [Chapter 3](#) for more on modules, [Chapter 2](#) for more on bytecode, and [Chapter 22](#) ahead for more on both. To really use the saved `.pyc` sans `.py`, as of Python 3.2, you must move it up one level and rename it without the `"*"` part in the middle, or generate it from and alongside the source code file with the Python `compileall` module's "legacy" (`-b`) mode. For example, the following compiles all source code files in the current directory into directly usable bytecode files (you can also list specific files or recurse into subfolders, per Python library docs):

```
$ python3 -m compileall -b -l .
```

4. *Scripts*: Assuming your platform supports the `#!` trick, your solution will look like [Example B-2](#), although your `#!` line may need to list a different path to Python on your machine. This line is significant under the Windows launcher shipped and installed with Python, where it is parsed to select a version of Python to run the script, despite the Unix path syntax, and subject to a default setting; see [Appendix A](#) and Python's docs for more details. This launching scheme is optional and generally less portable than others.

Example B-2. Part1/script1.py

```
#!/usr/local/bin/python3
print('Hello module world!')
```

Running this as a program by console command line:

```
$ chmod +x script1.py          # See also: #!/usr/bin/env python3
$ ./script1.py                 # "." needed only if "." not on PATH
Hello module world!

$ python3 script1.py           # Or run normally and portably
Hello module world!
```

5. *Errors and debugging*: The following interaction demonstrates the sorts of error messages you'll get when you complete this exercise. Really, you're triggering Python exceptions; the default exception-handling behavior terminates the running Python program and prints an error message and stack trace on the screen. The stack trace shows where you were in a program when the exception occurred (if function calls are active when the error happens, the "Traceback" section displays all active call levels).

In [Chapter 10](#) and [Part VII](#), you will learn that you can catch exceptions using `try` statements and process them arbitrarily. You'll also learn that Python includes a full-blown source code debugger (module `pdb`) for special error-detection requirements. For now, notice that Python gives meaningful messages when programming errors occur, instead of crashing silently:


```

$ python3
>>> 2 ** 500
32733906078961418700131896968275991522166420460430647894832913680961337964046745
54883270092325904157150886684127560071009217256545885393053328527589376

>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> oops
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'oops' is not defined

```

6. *Breaks and cycles*: When you type this code:

```

$ python3
>>> L = [1, 2]
>>> L.append(L)
>>> L
[1, 2, [...]]

```

you create a *cyclic* data structure in Python. In Python releases before 1.5.1, the Python printer wasn't smart enough to detect cycles in objects, and it would print an unending stream of `[1, 2, [1, 2, [1, 2, [1, 2, ...]]]` and so on until you hit the Ctrl+C break-key combination on your machine (which, technically, raises a keyboard-interrupt exception that prints a default message). Beginning with Python 1.5.1, the printer is clever enough to detect cycles, prints `[...]` instead to let you know that it has detected a loop in the object's structure, and avoids getting stuck printing forever.

The reason for the cycle is subtle and requires information you will glean in [Part II](#), so this is something of a preview. But in short, assignments in Python always generate *references* to objects, not copies of them. You can think of objects as chunks of memory and of references as implicitly followed pointers. When you run the first assignment in the preceding code, the name `L` becomes a named reference to a two-item list object—a pointer to a piece of memory. Python lists are really arrays of object references, with an `append` method that changes the array in place by tacking on another object reference at the end. Here, the `append` call adds a reference to the front of `L` at the end of `L`, which leads to the cycle illustrated in [Figure B-1](#): a pointer at the end of the list that points back to the front of the list.

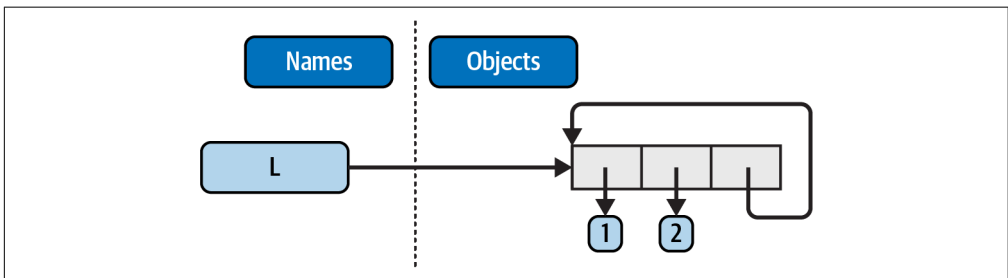


Figure B-1. A cyclic object, created by appending a list to itself

Besides being printed specially, as you'll learn in [Chapter 6](#), cyclic objects must also be handled specially by Python's garbage collector, or their space will remain unreclaimed even when they are

no longer in use. Though rare in practice, in some programs that traverse arbitrary objects or structures, you might have to detect such cycles yourself by keeping track of where you've been to avoid looping. Believe it or not, cyclic data structures can sometimes be useful, despite their special-case printing.

Part II, Objects and Operations

See “Test Your Knowledge: Part II Exercises” on page 239 in Chapter 9 for the exercises.

1. *The basics*: Here are the sorts of results you should get, along with a few comments about their meaning. Again, note that `;` is used in a few of these to squeeze more than one statement onto a single line (the `;` is a statement separator), and commas build up tuples displayed in parentheses. See file *Part2/basics.txt* for copy/paste sans emedia, though typing these manually is a good way to practice syntax:

```
$ python3
# Numbers

>>> 2 ** 16                                # 2 raised to the power 16
65536
>>> 2 / 5, 2 / 5.0                          # Division keep remainders
(0.4, 0.4)

# Strings

>>> 'hack' + 'code'                        # Concatenation
'hackcode'
>>> S = 'Python'
>>> 'grok ' + S
'grok Python'
>>> S * 5                                    # Repetition
'PythonPythonPythonPythonPython'
>>> S[0], S[:0], S[1:]                     # An empty slice at the front - [0:0]
('P', '', 'ython')                       # Empty of same type as object sliced

>>> how = 'fun'
>>> 'coding %s is %s!' % (S, how)          # Formatting: expression, method, f-string
'coding Python is fun!'
>>> 'coding {} is {}'.format(S, how)
'coding Python is fun!'
>>> f'coding {S} is {how}!'
'coding Python is fun!'

# Tuples

>>> ('x',)[0]                              # Indexing a single-item tuple
'x'
>>> ('x', 'y')[1]                          # Indexing a two-item tuple
'y'

# Lists

>>> L = [1, 2, 3] + [4, 5, 6]              # List operations
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])
>>> ([1, 2, 3] + [4, 5, 6])[2:4]
[3, 4]
>>> [L[2], L[3]]                          # Fetch from offsets; store in a list
```

```

[3, 4]
>>> L.reverse(); L           # Method: reverse list in place
[6, 5, 4, 3, 2, 1]
>>> L.sort(); L              # Method: sort list in place
[1, 2, 3, 4, 5, 6]
>>> L.index(4)               # Method: offset of first 4 (search)
3

# Dictionaries

>>> {'a': 1, 'b': 2}['b']     # Index a dictionary by key
2
>>> D = {'x': 1, 'y': 2, 'z': 3}
>>> D['w'] = 0                # Create a new entry
>>> D['x'] + D['w']
1
>>> D[(1, 2, 3)] = 4         # A tuple used as a key (immutable)

>>> D
{'x': 1, 'y': 2, 'z': 3, 'w': 0, (1, 2, 3): 4}

>>> list(D.keys()), list(D.values()), (1, 2, 3) in D      # Methods, key test
(['x', 'y', 'z', 'w', (1, 2, 3)], [1, 2, 3, 0, 4], True)

# Empties

>>> [], [], [], [], {}, None    # Lots of nothings: empty objects
([], [], [], [], {}, None)

```

2. *Indexing and slicing*: Indexing out of bounds (e.g., `L[4]`) raises an error; Python always checks to make sure that all offsets are within the bounds of a sequence.

On the other hand, slicing out of bounds (e.g., `L[-1000:100]`) works because Python scales out-of-bounds slices so that they always fit (the limits are set to zero and the sequence length, if required).

Extracting a sequence in reverse, with the lower bound greater than the higher bound (e.g., `L[3:1]`), doesn't really work. You get back an empty slice (`[]`) because Python scales the slice limits to make sure that the lower bound is always less than or equal to the upper bound (e.g., `L[3:1]` is scaled to `L[3:3]`, the empty insertion point at offset 3). Python slices are always extracted from left to right, even if you use negative indexes (they are first converted to positive indexes by adding the sequence length). Note that Python's three-limit slices modify this behavior somewhat. For instance, `L[3:1:-1]` does extract from right to left:

```

>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> L[-1000:100]
[1, 2, 3, 4]
>>> L[3:1]
[]

>>> L
[1, 2, 3, 4]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]

```

3. *Indexing, slicing, and del*: Your interaction with the interpreter should look something like the following. Note that assigning an empty list to an offset stores an empty list object there, but assigning an empty list to a slice deletes the slice. Slice assignment expects another sequence, or you'll get a type error; it inserts items *inside* the sequence assigned, not the sequence itself:

```
>>> L = [1, 2, 3, 4]
>>> L[2] = []
>>> L
[1, 2, [], 4]

>>> L[2:3] = []
>>> L
[1, 2, 4]

>>> del L[0]
>>> L
[2, 4]
>>> del L[1:]
>>> L
[2]

>>> L[1:2] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
```

4. *Tuple assignment*: The values of *X* and *Y* are swapped. When tuples appear on the left and right of an assignment symbol (=), Python assigns objects on the right to targets on the left according to their positions. This is probably easiest to understand by noting that the targets on the left aren't a real tuple, even though they look like one; they are simply a set of independent assignment targets. The items on the right are a tuple, which gets unpacked during the assignment (this tuple provides the temporary assignment needed to achieve the swap effect):

```
>>> X = 'code'
>>> Y = 'hack'
>>> X, Y = Y, X
>>> X
'hack'
>>> Y
'code'
```

5. *Dictionary keys*: Any *immutable* (technically, “hashable”) object can be used as a dictionary key, including integers, tuples, strings, and so on. This really is a dictionary, even though some of its keys look like integer offsets. Mixed-type keys work fine, too:

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. *Dictionary indexing*: Indexing a nonexistent key (`D['d']`) raises an error; assigning to a nonexistent key (`D['d']='hack'`) creates a new dictionary entry. On the other hand, out-of-bounds indexing for lists raises an error, too, but so do out-of-bounds assignments. Variable names work like dictionary keys; they must have already been assigned when referenced, but they are created when first assigned. In fact, variable names can be processed as dictionary keys if you wish (they're made visible in the dictionaries of stack frames or module [or other object] namespaces):

```

>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D['a']
1
>>> D['d']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'd'

>>> D['d'] = 4
>>> D
{'a': 1, 'b': 2, 'c': 3, 'd': 4}

>>> L = [0, 1]
>>> L[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> L[2] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range

```

7. *Generic operations:* Question answers (with some error text omitted in listings):

- a. The + operator doesn't work on different/mixed types (e.g., string + list, list + tuple).
- b. + doesn't work for dictionaries, as they aren't sequences (though | does).
- c. The append method works only for lists, not strings, and keys works only on dictionaries. append assumes its target is mutable, since it's an in-place extension; strings are immutable. Dictionary keys is similarly type specific.
- d. Slicing and concatenation always return a new object of the same type as the objects processed:

```

>>> 'x' + 1
TypeError: illegal argument type for built-in operation

>>> {} + {}
TypeError: bad operand type(s) for +

>>> [].append(9)
>>> ''.append('s')
AttributeError: attribute-less object

>>> list({}.keys())
[]
>>> [].keys()
AttributeError: keys

>>> [][::]
[]
>>> ''[::]
''

```

8. *String indexing:* This is a bit of a trick question—because strings are collections of one-character strings, every time you index a string, you get back a string that can be indexed again. `S[0][0][0][0][0]` just keeps indexing the first character over and over. This generally doesn't work for lists (lists can hold arbitrary objects) unless the list contains strings:

```
>>> S = 'hack'
>>> S[0][0][0][0][0]
'h'
>>> L = ['h', 'a']
>>> L[0][0][0]
'h'
```

9. *Immutable types*: Either of the following solutions works. Index assignment doesn't because strings are immutable:

```
>>> S = 'hack'
>>> S = S[0] + 'e' + S[2:]
>>> S
'heck'
>>> S = S[0] + 'i' + S[2] + S[3]
>>> S
'hick'
```

(See also the bytearray string type in [Chapter 37](#)—it's a mutable sequence of small integers that is essentially processed the same as a string, especially when its bytes are ASCII character code points.)

10. *Nesting*: Here is a sample (your specs will vary):

```
>>> pat = {'name': ('Pat', 'Q', 'Jones'), 'age': None, 'job': 'engineer'}
>>> pat['job']
'engineer'
>>> pat['name'][2]
'Jones'
```

11. *Files*: Examples B-3 and B-4 show one way to create and read back a text file in Python using Unicode encoding defaults on the host (which are generally moot for simple ASCII text like this):

Example B-3. Part2/maker.py

```
file = open('myfile.txt', 'w')
file.write('Hello file world!\n')
file.close()
```

Or: open().write()
close not always needed

Example B-4. Part2/reader.py

```
file = open('myfile.txt')
print(file.read())
```

'r' is default open mode
Or print(open().read())

When run (here, from a console command line), the file shows up in the directory you're working in because its name has no path prefix. The `ls` here is a Unix command; use `dir` on Windows:

```
$ python3 maker.py
$ python3 reader.py
Hello file world!

$ ls -l myfile.txt
-rw-r--r--  1 me  staff  18 Aug 11 19:34 myfile.txt
```

Part III, Statements and Syntax

See “Test Your Knowledge: Part III Exercises” on page 374 in Chapter 15 for the exercises.

1. *Coding basic loops:* As you work through this exercise, you’ll wind up with code that looks like the following:

```
>>> S = 'hack'
>>> for c in S:
...     print(ord(c))
...
104
97
99
107

>>> x = 0
>>> for c in S: x += ord(c)           # Or: x = x + ord(c)
...
>>> x
407
>>> chr(x)                           # Extra credit: non-ASCII, see Chapter 37
'I'

>>> x = []
>>> for c in S: x.append(ord(c))      # Manual list construction
...
>>> x
[104, 97, 99, 107]

>>> list(map(ord, S))
[115, 112, 97, 109]
>>> [ord(c) for c in S]               # map and listcomps automate list builders
[115, 112, 97, 109]
```

2. *Coding basic selections:* Here is the sort of code expected. To handle out-of-range numbers, add an else for if, a case _ for match, a get method call or in test for the dictionary, and a try handler for the list. For versions of this code that are easier to copy/paste, see file *Part3/selections.txt* in the examples package:

```
>>> month = 3
>>> if month == 1:
...     print('January')
... elif month == 2:
...     print('February')
... elif month == 3:
...     print('March')
...
March

>>> match month:
...     case 1:
...         print('January')
...     case 2:
...         print('February')
...     case 3:
...         print('March')
...
March
```

```
>>> {1: 'January', 2: 'February', 3: 'March'}[month]
'March'
>>> ['January', 'February', 'March'][month - 1]
'March'
```

3. *Backslash characters*: The example prints the bell character (`\a`) 50 times. Assuming your machine can handle it, and when it's run outside of some interfaces like IDLE, you may get a series of beeps (or one sustained tone if your machine is fast enough). Hey—you were warned.
4. *Sorting dictionaries*: Here's one way to work through this exercise (see [Chapter 8](#) or [Chapter 14](#) if this doesn't make sense). You really do have to split off the keys and `sort` calls like this because `sort` returns `None`. You can iterate through dictionary keys directly without calling `keys` (e.g., for `key in D:`), but the keys list will not be sorted like it is by this code. The sorted built-in is simpler but creates a new list object:

```
>>> D = {'a': 1, 'c': 3, 'e': 5, 'g': 7, 'f': 6, 'd': 4, 'b': 2}
>>> D
{'a': 1, 'c': 3, 'e': 5, 'g': 7, 'f': 6, 'd': 4, 'b': 2}

>>> keys = list(D.keys())           # Keys view has no sort method
>>> keys.sort()                     # Sort list in place: returns None
>>> for key in keys:                 # Iterate over sorted list
...     print(key, '=>', D[key])
...
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
g => 7

>>> D
{'a': 1, 'c': 3, 'e': 5, 'g': 7, 'f': 6, 'd': 4, 'b': 2}
>>>
>>> for key in sorted(D):             # Simpler alternative, but a new list
...     print(key, '=>', D[key])
...
...same output...
```

5. *Program logic alternatives*: Here's some sample code for the solutions, available in the examples package's `Part3/power*.py`. For step e, assign the result of `2 ** X` to a variable outside the loops of steps a and b and use it inside the loop. Your results may vary; this exercise is mostly designed to get you playing with code alternatives, so anything reasonable gets full credit:

```
# a

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

i = 0
while i < len(L):
    if 2 ** X == L[i]:
        print('at index', i)
        break
    i += 1
else:
    print(X, 'not found')

# b
```



```

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

for p in L:
    if (2 ** X) == p:
        print((2 ** X), 'was found at', L.index(p))
        break
    else:
        print(X, 'not found')

# c

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# d

X = 5
L = []
for i in range(7): L.append(2 ** i)
print(L)

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# "Deeper thoughts"

X = 5
L = list(map(lambda x: 2 ** x, range(7)))    # Or [2 ** x for x in range(7)]
print(L)

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

```

Part IV, Functions and Generators

See “Test Your Knowledge: Part IV Exercises” on page 554 in Chapter 21 for the exercises.

1. *The basics:* There’s not much to this one, but notice that using `print` (and hence your function) is technically a *polymorphic* operation, which does the right thing for each type of object:

```

$ python3
>>> def echo(x):
        print(x)

>>> echo('hack')
hack
>>> echo(3.12)

```

```

3.12
>>> echo([1, 2, 3])
[1, 2, 3]
>>> echo({'edition': 6})
{'edition': 6}

```

2. *Arguments:* [Example B-5](#) gives a sample solution. Remember that you have to use `print` to see results in the test calls because a file isn't the same as code typed interactively; Python doesn't normally echo the results of expression statements in files:

Example B-5. Part4/adder1.py

```

def adder(x, y):
    return x + y

print(adder(5, 1.0))
print(adder('hack', 'code'))
print(adder(['a', 'b'], ['c', 'd']))

```

And the output:

```

$ python3 adder1.py
6.0
hackcode
['a', 'b', 'c', 'd']

```

3. *Arbitrary arguments:* Two alternative `adder` functions are shown in [Example B-6](#). The hard part here is figuring out how to initialize an accumulator to an empty value of whatever type is passed in. The first solution uses manual type testing to look for an integer and an empty slice of the first argument (assumed to be a sequence) if the argument is determined not to be an integer. The second solution uses the first argument to initialize and scan items 2 and beyond, much like one of the `min` function variants shown in [Chapter 18](#).

The second solution may be better. Both of these assume all arguments are of the same type, and neither works on dictionaries (as we saw in [Part II](#), `+` doesn't work on mixed types or dictionaries). You could add a type test and special code using `for`, `update`, `**`, or `|` to support dictionaries combos, too, but that's extra credit; see solutions 5 and 6 ahead for related notes. And yes, there is a `sum(iterable)` built-in in Python that would make this even simpler, but the point here is to write code of your own; you'll have to eventually:

Example B-6. Part4/adder2.py

```

def adder1(*args):
    print('adder1:', end=' ')
    if type(args[0]) == type(0):
        sum = 0
    else:
        sum = args[0][:0]
    for arg in args:
        sum = sum + arg
    return sum

def adder2(*args):
    print('adder2:', end=' ')
    sum = args[0]
    for next in args[1:]:
        sum += next
    return sum

```

```

# Integer?
# Init to zero
# else sequence:
# Use empty slice of arg1

# Init to arg1
# Add items 2..N

```

```
for func in (adder1, adder2):
    print(func(2, 3, 4))
    print(func('hack', 'code', 'well'))
    print(func(['a', 'b'], ['c', 'd'], ['e', 'f']))
```

Here's the sort of output you should get:

```
$ python3 adder2.py
adder1: 9
adder1: hackcodewell
adder1: ['a', 'b', 'c', 'd', 'e', 'f']
adder2: 9
adder2: hackcodewell
adder2: ['a', 'b', 'c', 'd', 'e', 'f']
```

4. **Keywords:** [Example B-7](#) gives a solution to the first part of this exercise, along with its output in a console.

Example B-7. Part4/adder3.py

```
def adder(red=1, green=2, blue=3):
    return red + green + blue

print(adder())
print(adder(5))
print(adder(5, 6))
print(adder(5, 6, 7))
print(adder(blue=7, red=6, green=5))
print(adder(blue=1, red=2))
```

```
$ python3 adder3.py
6
10
14
18
18
5
```

Example B-8 gives the second part's solution and its output. To iterate over keyword arguments, use the `**args` form in the function header and use a loop (e.g., `for x in args.keys():` use `args[x]`), or use `args.values()` to make this the same as summing `*args` positionals in exercise number 3:

Example B-8. Part4/adder4.py

```
def adder1(*args):                                # Sum any number of positional args
    tot = args[0]                                  # Same as #3, for comparison and reuse
    for arg in args[1:]:
        tot += arg
    return tot

def adder2(**args):                                # Sum any number of keyword args
    argskeys = list(args.keys())                   # list required to index!
    tot = args[argskeys[0]]
    for key in argskeys[1:]:
        tot += args[key]
    return tot
```

```

def adder3(**args):                # Same, but convert to list of values
    args = list(args.values())     # list needed to index!
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder4(**args):                # Same, but reuse positional version
    return adder1(*args.values())

print(adder1(1, 2, 3),          adder1('aa', 'bb', 'cc'))
print(adder2(a=1, b=2, c=3), adder2(a='aa', b='bb', c='cc'))
print(adder3(a=1, b=2, c=3), adder3(a='aa', b='bb', c='cc'))
print(adder4(a=1, b=2, c=3), adder4(a='aa', b='bb', c='cc'))

$ python3 adder4.py
6 aabbcc
...repeated 4 times...

```

5. (5 and 6) *Dictionary tools*: Solutions for exercises 5 and 6 are combined and listed in [Example B-9](#). These are just coding exercises because Python now provides dictionary methods `D.copy()` and `D1.update(D2)` to handle things like copying and adding (merging) dictionaries. In fact, there are *four* ways to merge dictionaries today, as hinted in solution 3: for loops like those here, `D1.update(D2)`, `{**D1,**D2}`, and `D1|D2`. See [Chapter 8](#) for more info on and examples of these tools. `X[:]` doesn't work for dictionaries, as they're not sequences (see [Chapter 8](#) for details). Also, remember that if you assign (`e = d`) rather than copying, you generate a reference to a *shared* dictionary object; changing `d` changes `e`, too:

Example B-9. Part4/dicttools.py

```

def copyDict(old):
    new = {}
    for key in old.keys():
        new[key] = old[key]
    return new

def addDict(d1, d2):
    new = {}
    for key in d1.keys():
        new[key] = d1[key]
    for key in d2.keys():
        new[key] = d2[key]
    return new

```

Here is the expected behavior of this code demoed in a REPL:

```

$ python3
>>> from dicttools import *
>>> d = {1: 1, 2: 2}
>>> e = copyDict(d)
>>> d[2] = '?'
>>> d
{1: 1, 2: '?'}
>>> e
{1: 1, 2: 2}

>>> x = {1: 1}
>>> y = {2: 2}

```

```
>>> z = addDict(x, y)
>>> z
{1: 1, 2: 2}
```

6. See #5 (where solutions were combined).
7. *More argument-matching examples:* Here is the sort of interaction you should get, along with comments that explain the matching that goes on. It may be easiest to paste the functions into a file and import them all with a `*` for testing in a REPL; they're repeated in [Example B-10](#) for reference (and in the examples package for copying):

Example B-10. Part4/testfuncs.py

```
def f1(a, b): print(a, b)           # Normal args
def f2(a, *b): print(a, b)         # Positional collectors
def f3(a, **b): print(a, b)        # Keyword collectors
def f4(a, *b, **c): print(a, b, c) # Mixed modes
def f5(a, b=2, c=3): print(a, b, c) # Defaults
def f6(a, b=2, *c): print(a, b, c) # Defaults and positional collector
```

The expected REPL interaction:

```
$ python3
>>> from testfuncs import *
>>> f1(1, 2)                      # Matched by position (order matters)
1 2
>>> f1(b=2, a=1)                  # Matched by name (order doesn't matter)
1 2

>>> f2(1, 2, 3)                   # Extra positionals collected in a tuple
1 (2, 3)

>>> f3(1, x=2, y=3)               # Extra keywords collected in a dictionary
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, **dict(x=2, y=3)) # Extras of both kinds, star unpacking
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1)                         # Both defaults kick in
1 2 3
>>> f5(1, 4)                      # Only one default used
1 4 3
>>> f5(1, c=4)                   # Middle default applied
1 2 4

>>> f6(1)                         # One argument: matches "a"
1 2 ()
>>> f6(1, *[3, 4])               # Extra positional collected, star unpacking
1 3 (4,)
```

8. *Primes revisited*: **Example B-11** is the primes example, wrapped up in a function and a module (file `primes.py`) so it can be run multiple times. An `if` test was added to trap negatives, 0, and 1. It's crucial to use `//` floor division instead of the `/` true division we studied in **Chapter 5** to avoid fractional remainders (`5 / 2` would yield a false factor 2.5, but `5 // 2` truncates down to 2). Change `//` to `/` to see the difference for yourself:

Example B-11. Part4/primes.py

```
def prime(y):
    if y <= 1:                                # For some y > 1
        print(y, 'is nonprime')
    else:
        x = y // 2                            # But / fails
        while x > 1:
            if y % x == 0:                    # No remainder?
                print(y, 'has factor', x)
                break                        # Skip else
            x -= 1
        else:
            print(y, 'is prime')

tests = (27, 24, 13, 13.0, 15, 15.0, 3, 2, 1, -3)
for test in tests:
    prime(test)
```

Here is the module in action; the `//` operator also allows it to work for floating-point numbers by truncating to the floor (`5.0 // 2` is 2.0, not 2.5):

```
$ python3 primes.py
27 has factor 9
24 has factor 12
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
3 is prime
2 is prime
1 is nonprime
-3 is nonprime
```

This function still isn't very reusable—it could *return* values, instead of printing—but it's enough to run experiments. It's also not a strict mathematical prime (floating-point numbers work, but shouldn't), and it's still perhaps inefficient. Improvements are left as exercises for more mathematically minded readers. (Hint: a `for` loop over `range(x, 1, -1)` may be a bit quicker than the `while`, but the algorithm may be the real bottleneck here.) To time alternatives, use the homegrown `timer` or standard-library `timeit` modules and coding patterns like those used in **Chapter 21**'s benchmarking sections (and solution 10 ahead).

9. *Iterations and comprehensions*: Here is the sort of code you should write; coding alternatives are notoriously subjective, so there's no right or wrong preference (though see the next solution for an objective factor):

```
>>> values = [2, 4, 9, 16, 25]
>>> import math

>>> res = []
>>> for x in values: res.append(math.sqrt(x))    # Manual loop
...

```

```
>>> res
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> list(map(math.sqrt, values))           # map built-in call
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> [math.sqrt(x) for x in values]         # List comprehension
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> list(math.sqrt(x) for x in values)     # Generator expression
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
```

10. *Timing tools*: The code file in [Example B-13](#) times the three square root options. Each test takes the best of 5 runs; each run takes the total time required to call the test function 1,000 times; and each test function iterates 10,000 times. The last result of each function is printed to verify that all three do the same work.

This code also uses a preview (really, cheat) to remotely access the *timer2.py* module in Chapter 21's code folder ([Example B-12](#)) with an `import` run its own folder, assumed to be the examples' *AppendixB/Part4*. Appending `sys.path` is one way to augment the search path used to find imported modules, along with `PYTHONPATH` environment settings. This avoids a file copy; we'll explore it in this book's next part, so take it on faith for now.

Example B-12. .../Chapter21/timer2.py

...Example 21-7 in Chapter 21...

Example B-13. Part4/timesqrt.py

```
import sys                                # Add timer2.py's folder to search path
sys.path.append('../Chapter21')          # Assuming running in AppendixB/Part4
import timer2                             # A cheat! - see Part V for path info

reps = 10_000
repslist = list(range(reps))             # Pull out range list time

from math import sqrt                     # Not math.sqrt: adds attr fetch time
def mathMod():
    for i in repslist:
        res = sqrt(i)
    return res

def powCall():
    for i in repslist:
        res = pow(i, .5)
    return res

def powExpr():
    for i in repslist:
        res = i **.5
    return res

print(sys.version)
for test in (mathMod, powCall, powExpr):
    elapsed, result = timer2.bestoftotal(test, _reps1=5, _reps=1000)
    print(f'{test.__name__}: {elapsed:.5f} => {result}')
```

Following are the test results for CPython 3.12 (the standard) and PyPy 7.3 (which implements Python 3.10) on macOS. In short, the `math` module is quicker than the `**` expression on both Pythons, and `**` is quicker than the `pow` built-in function in CPython but the same in PyPy:

```
$ python3 timesqrt.py
3.12.2 (v3.12.2:6abddd9f6a, Feb 6 2024, 17:02:06) [Clang 13.0.0 (clang-1300.0.29.30)]
mathMod: 0.40860 => 99.99499987499375
powCall: 0.68245 => 99.99499987499375
powExpr: 0.57762 => 99.99499987499375

$ pypy3 timesqrt.py
3.10.14 (75b3de9d9035, Apr 21 2024, 10:56:19)
[PyPy 7.3.16 with GCC Apple LLVM 15.0.0 (clang-1500.1.0.2.5)]
mathMod: 0.05246 => 99.99499987499375
powCall: 0.33288 => 99.99499987499375
powExpr: 0.33244 => 99.99499987499375
```

PyPy is also some 8X to 2X faster than CPython on floating-point math and iterations here, but CPython may sprout a JIT, which evens the gap (see [Chapter 2](#)). The results for CPython jive with the prior edition's tests for CPython 3.3 that follow, which used different repeat counts and hosts but were relatively similar. As always, you should try this with your code and on your own machine and version of Python for more definitive results:

```
c:\code> py -3 timesqrt.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
mathMod: 2.04481 => 99.99499987499375
powCall: 3.40973 => 99.99499987499375
powExpr: 2.56458 => 99.99499987499375
```

To time the relative speeds of *dictionary comprehensions* and equivalent for loops interactively, you can run a session like the following. At least on this test in CPython 3.12, the two are roughly the same in speed, with a slight advantage to comprehensions—though the difference isn't exactly earth-shattering. As verification, these results relatively match those we obtained from a `pybench` test in [Example 21-10](#) (sans the slower `dict` call). Do similar to vet the speed of comprehensions with `if` and `for`. And again, rather than taking any of these results as gospel, you should investigate further on your own with your computer and your Python:

```
$ python3
>>> def dictcomp(I):
    return {i: i for i in range(I)}

>>> def dictloop(I):
    new = {}
    for i in range(I): new[i] = i
    return new

>>> dictcomp(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>> dictloop(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}

>>> import sys; sys.path.append('.././Chapter21')
>>> from timer2 import bestoftotal

>>> bestoftotal(dictcomp, 10_000, _reps1=5, _reps=500)[0]
0.17137739405734465
>>> bestoftotal(dictloop, 10_000, _reps1=5, _reps=500)[0]
0.18112968490459025
```



```
>>> len(bestoftotal(dictcomp, 10_000, _reps1=5, _reps=500)[1])
10000
>>> len(bestoftotal(dictloop, 10_000, _reps1=5, _reps=500)[1])
10000
```

11. *Recursive functions*: One way to code this function follows (typed in a REPL here, but also coded in file *Part4/countdown.py* of the examples package). A simple range, comprehension, or map will do the job here as well, of course, but recursion is useful enough to warrant the experimentation here:

```
>>> def countdown(N):
    if N == 0:
        print('stop')
    else:
        print(N, end=' ')
        countdown(N - 1)

>>> countdown(5)
5 4 3 2 1 stop
>>> countdown(20)
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 stop

# Nonrecursive options

>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
>>> t = [print(i, end=' ') for i in range(5, 0, -1)]
5 4 3 2 1
>>> t = list(map(lambda x: print(x, end=' '), range(5, 0, -1)))
5 4 3 2 1
```

A *generator*-based solution isn't required for this exercise, but one is listed next; all the other techniques seem much simpler in this case—a good example of contexts where generators should probably be avoided. Remember that generators produce no results until iterated, so we need a for loop or yield from here (yielding `countdown2(N-1)` directly simply returns a generator, not its products):

```
>>> def countdown2(N):                                     # Generator function, recursive
    if N == 0:
        yield 'stop'
    else:
        yield N
        for x in countdown2(N - 1): yield x                # Or: yield from countdown2(N - 1)

>>> list(countdown2(5))
[5, 4, 3, 2, 1, 'stop']

# Nonrecursive options

>>> def countdown3():                                     # Generator function, simpler
    yield from range(5, 0, -1)                             # Or: for x in range(): yield x

>>> list(countdown3())
[5, 4, 3, 2, 1]
>>> list(x for x in range(5, 0, -1))                     # Equivalent generator expression
[5, 4, 3, 2, 1]
```

12. *Computing factorials*: [Example B-14](#) shows one way to code this exercise, using Python's standard-library `timeit` module of [Chapter 21](#). Naturally, there are many possible variations on its code; its ranges, for instance, could run from `2..N+1` to skip an iteration, and `fact2` could use `reduce(operator.mul, range(N, 1, -1))` to avoid a `lambda`. Improve freely.

Example B-14. Part4/factorials.py

```
from functools import reduce
from timeit import repeat
import math

def fact0(N):
    if N == 1:
        return N
    else:
        return N * fact0(N - 1)
    # Recursive
    # Fails at stack limit

def fact1(N):
    return N if N == 1 else N * fact1(N - 1)
    # Recursive, one-liner

def fact2(N):
    return reduce(lambda x, y: x * y, range(1, N + 1))
    # Functional

def fact3(N):
    res = 1
    for i in range(1, N + 1): res *= i
    return res
    # Iterative

def fact4(N):
    return math.factorial(N)
    # Stdlib "batteries"

# Tests
print(fact0(6), fact1(6), fact2(6), fact3(6), fact4(6))    # 6*5*4*3*2*1: all 720
print(fact0(500) == fact1(500) == fact2(500) == fact3(500) == fact4(500)) # True

for test in (fact0, fact1, fact2, fact3, fact4):
    print(test.__name__, min(repeat(stmt=lambda: test(500), number=1000, repeat=5)))
```

This code uses Python's `timeit` module to benchmark alternatives. Its results for CPython 3.12 on macOS:

```
$ python3 factorials.py
720 720 720 720 720
True
fact0 0.08720566902775317
fact1 0.08635473699541762
fact2 0.06704489700496197
fact3 0.05152398400241509
fact4 0.00873392098583281
```

Conclusions: recursion is slowest on this Python and machine and fails once `N` reaches the maximum stack-size setting in `sys`. Per [Chapter 19](#), this limit can be increased, but simple loops or the standard-library tool seem the best route here in any event, and the built-in wins soundly.

This general finding holds true often. For instance, `''.join(reversed(S))` may be the preferred way to reverse a string, even though recursive solutions are possible. Time the code in [Example B-15](#) to see for yourself:

Example B-15. *Part4/reverses.py*

```
def rev1(S):
    if len(S) == 1:
        return S
    else:
        return S[-1] + rev1(S[:-1])    # Recursive

def rev2(S):
    return ''.join(reversed(S))        # Nonrecursive iterable

def rev3(S):
    return S[::-1]                    # Sequence reversal by slice
```

Part V, Modules and Packages

See “Test Your Knowledge: Part V Exercises” on page 644 in Chapter 25 for the exercises.

1. *Import basics*: When you’re done, your file and REPL interaction with it should look similar to [Example B-16](#). Remember that Python can read a whole file into a list of line strings, and the `len` built-in returns the lengths of strings and lists:

Example B-16. *Part5/mymod.py (initial code, mymod_start.py)*

```
def countLines(name):
    file = open(name)
    return len(file.readlines())

def countChars(name):
    return len(open(name).read())

def test(name):
    return countLines(name), countChars(name)    # Or pass file object
                                                # Or return a dictionary

$ python3
>>> import mymod
>>> mymod.test('mymod.py')
(10, 281)
```

Your counts may vary for comments, an extra line at the end, and so on, and you don’t need to set `PYTHONPATH` if the module is in the automatically searched current working directory. Note that these functions load the entire file in memory all at once, so they won’t work for pathologically large files that are too big for your device’s memory. To be more robust, you could read line by line with iterators instead and count as you go (see *Part5/mymod_lines.py* in the examples package):

```
def countLines(name):
    tot = 0
    for line in open(name): tot += 1
    return tot

def countChars(name):
    tot = 0
    for line in open(name): tot += len(line)
    return tot
```

A generator expression can have the same effect (though the excessive magic may cost you some points):

```
def countLines(name): return sum(+1 for line in open(name))
def countChars(name): return sum(len(line) for line in open(name))
```

On Unix, you can verify your output with a `wc` command; on Windows, right-click on your file to view its properties. Note that your script may report fewer characters than Windows does—for portability, Python converts Windows `\r\n` line-end markers to `\n`, thereby dropping one byte (character) per line. To match byte counts with Windows exactly, you must open in binary mode (`'rb'`) or add the number of bytes corresponding to the number of lines. See Chapters 9 and 37 for more on end-of-line translations in text files.

The “ambitious” part of this exercise (passing in a file object so you only open the file once) will require you to use the `seek` method of the built-in file object. It works like C’s `fseek` call (and may call it behind the scenes): `seek` resets the current position in the file to a passed-in offset. After a `seek`, future input/output operations are relative to the new position. To rewind to the start of a file without closing and reopening it, call `file.seek(0)`; the file `read` methods all pick up at the current position in the file, so you need to rewind to reread. Example B-17 shows what this tweak would look like, along with its output in a REPL:

Example B-17. Part5/mymod2.py

```
def countLines(file):
    file.seek(0)                                # Rewind to start of file
    return len(file.readlines())

def countChars(file):
    file.seek(0)                                # Ditto (rewind if needed)
    return len(file.read())

def test(name):
    file = open(name)                           # Pass file object
    return countLines(file), countChars(file)    # Open file only once

$ python3
>>> import mymod2
>>> mymod2.test('mymod2.py')
(12, 414)
```

2. `from/from *`: Here’s the `from *` part; replace `*` with `countChars` to do the rest:

```
$ python3
>>> from mymod import *
>>> countChars('mymod.py')
281
```

3. `__main__`: If you code it properly, this file works in either mode—program run or module import, as Example B-18 and the REPL session following it demo:

Example B-18. Part5/mymod.py (edited)

```
def countLines(name):
    file = open(name)
    return len(file.readlines())

def countChars(name):
    return len(open(name).read())
```

```

def test(name):
    return countLines(name), countChars(name)
# Or pass file object
# Or return a dictionary

if __name__ == '__main__':
    print(test('mymod.py'))
# Added: self-test code
# When run, not when imported

$ python3 mymod.py
(13, 434)

```

This is where you would probably begin to consider using command-line arguments or user input to provide the filename to be counted instead of hardcoding it in the script. Examples B-19 and B-20 show the required mods (see Chapters 21 and 25 for more on `sys.argv`, and Chapter 10 for more on input):

Example B-19. Part5/mymod_argv.py (changed parts)

```

...
if __name__ == '__main__':
    import sys
    print(test(sys.argv[1]))
# Command-line argument

$ python3 mymod_argv.py mymod.py
(13, 434)

```

Example B-20. Part5/mymod_input.py (changed parts)

```

...
if __name__ == '__main__':
    print(test(input('Enter file name: ')))
# Console/user input

$ python3 mymod_input.py
Enter file name: mymod.py
(13, 434)

```

4. *Nested imports*: It's not much, but Example B-21 gives one solution and its results (the point here is to experiment with importing one module from another in a variety of ways):

Example B-21. Part5/myclient.py

```

from mymod import countLines, countChars
print(countLines('mymod.py'), countChars('mymod.py'))

$ python3 myclient.py
13 434

```

As for the rest of this question, `mymod`'s functions are accessible (that is, importable) from the top level of `myclient`, since `from` simply assigns to names in the importer (it works as if `mymod`'s defs appeared in `myclient`). For example, another file can say:

```

import myclient
myclient.countLines(...)

from myclient import countChars
countChars(...)

```

If `myclient` used `import` instead of `from`, you'd need to use a path to get to the functions in `mymod` through `myclient`:

```
import myclient
myclient.mymod.countLines(...)

from myclient import mymod
mymod.countChars(...)
```

In general, you can define *collector* modules that import all the names from other modules so they're available in a single convenience module. The following hypothetical code, for example, creates three different copies of the name `somename`—`mod1.somename`, `collector.somename`, and `__main__.somename`; all three share the same integer object initially, and only the name `somename` exists at the interactive prompt as is:

```
# File mod1.py (hypothetical)
somename = 99

# File collector.py (hypothetical)
from mod1 import *           # Collect lots of names here
from mod2 import *           # "from" assigns to my names
from mod3 import *

>>> from collector import somename
```

5. **Package imports:** For this, copy the `mymod.py` solution file listed for exercise 3 (Example B-18) into a directory package. The following commands run in a Unix console set up the directory and an optional `__init__.py` file; you'll need to interpolate for other platforms and tools (e.g., use `copy` and `notepad` on Windows instead of `cp` and `vi`). This works in any directory, and you can do some of this from a file-explorer GUI, too.

When finished, you'll have a `mypkg` subdirectory that contains the files `__init__.py` and `mymod.py`. Technically, `mypkg` is located in the “home” directory component of the module search path. Notice how a `print` statement coded in the directory's initialization file fires only the first time it is imported, not the second. Raw strings (`r'...'`) can also avoid `\` escape issues in the file paths if you're working on Windows, but `/` works there too:

```
$ mkdir mypkg                               # Windows: same
$ cp mymod.py mypkg/mymod.py                # Windows: copy mymod.py mypkg\mymod.py
$ vi mypkg/__init__.py                       # Windows: notepad mypkg\__init__.py
...code a print statement...

$ python3                                    # Windows: py -3 (probably)
>>> import mypkg.mymod
initializing mypkg
>>> mypkg.mymod.countLines('mypkg/mymod.py') # Windows: same
13
>>> from mypkg.mymod import countChars
>>> countChars('mypkg/mymod.py')             # Windows: same
434
```

If you copy the module to `__main__.py`, the copy will run if you run the directory as a whole (though there may be no reason to do so in practice, as the original module can be run directly too):

```
$ cp mypkg/mymod.py mypkg/__main__.py       # Windows: copy
$ python3 mypkg
(13, 434)
```

```
$ python3 mypkg/mymod.py
(13, 434)
```

6. *Reloads*: This exercise just asks you to experiment with changing the *changer.py* example in the book's [Example 23-10](#), so there's nothing to show here.
7. *Circular imports*: The short story is that importing *recur2* first works because the recursive import then happens at the import in *recur1*, not at a *from* in *recur2*.

The long story goes like this: importing *recur2* first works because the recursive import from *recur1* to *recur2* fetches *recur2* as a whole instead of getting specific names. *recur2* is incomplete when it's imported from *recur1*, but because it uses `import` instead of `from`, you're safe: Python finds and returns the already created *recur2* module object and continues to run the rest of *recur1* without a glitch. When the *recur2* import resumes, the second *from* finds the name `Y` in *recur1* (it's been run completely), so no error is reported.

Running a file as a *script* is not the same as importing it as a module; these cases are the same as running the first `import` or `from` in the script interactively. For instance, running *recur1* as a script works because it is the same as importing *recur2* interactively, as *recur2* is the first module imported in *recur1*. Running *recur2* as a script fails for the same reason—it's the same as running its first import interactively.

Part VI, Classes and OOP

See “[Test Your Knowledge: Part VI Exercises](#)” on page 866 in [Chapter 32](#) for the exercises.

1. *Inheritance*: [Example B-22](#) lists a solution for this exercise, along with some interactive tests. The `__add__` overload has to appear only once, in the superclass, as it invokes type-specific `add` methods in subclasses:

Example B-22. Part6/adder.py

```
class Adder:
    def add(self, x, y):
        print('not implemented!')
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):
        return self.add(self.data, other)

class ListAdder(Adder):
    def add(self, x, y):
        return x + y

class DictAdder(Adder):
    def add(self, x, y):
        new = {}
        for k in x.keys(): new[k] = x[k]
        for k in y.keys(): new[k] = y[k]
        return new

$ python3
>>> from adder import *
>>> x = Adder()
>>> x.add(1, 2)
not implemented!
```

```

>>> x = ListAdder()
>>> x.add([1], [2])
[1, 2]
>>> x = DictAdder()
>>> x.add({1: 1}, {2: 2})
{1: 1, 2: 2}

>>> x = Adder([1])
>>> x + [2]
not implemented!
>>>
>>> x = ListAdder([1])
>>> x + [2]
[1, 2]
>>> [2] + x
TypeError: can only concatenate list (not "ListAdder") to list

```

Notice in the last test that you get an error for expressions where a class instance appears on the right of a +; if you want to fix this, use `__radd__` methods, as described in [Chapter 30](#).

If you are saving a value in the instance anyhow, you might as well rewrite the `add` method to take just one argument, in the spirit of other examples in this part of the book. [Example B-23](#) sketches this mutation:

Example B-23. Part6/adder2.py

```

class Adder:
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):
        return self.add(other)
    def add(self, y):
        print('not implemented!')

class ListAdder(Adder):
    def add(self, y):
        return self.data + y

class DictAdder(Adder):
    def add(self, y):
        d = self.data.copy()
        d.update(y)
        return d

x = ListAdder([1, 2, 3])
y = x + [4, 5, 6]
print(y)

z = DictAdder(dict(name='x')) + {'a': 1}
print(z)

```

Because values are attached to objects rather than passed around, this version is arguably more object-oriented. And, once you've gotten to this point, you'll probably find that you can get rid of `add` altogether and simply define type-specific `__add__` methods in the two subclasses.

2. *Operator overloading*: The solution code and its REPL results in [Example B-24](#) demo a handful of operator-overloading methods we explored in [Chapter 30](#). Copying the initial value in the constructor is important because it may be mutable; you don't want to change or have a reference to an

object that's possibly shared somewhere outside the class. The `__getattr__` method routes calls to the wrapped list. For tips on a possibly easier way to code this, See “[Extending Types by Subclassing](#)” on page 819 in [Chapter 32](#):

Example B-24. *Part6/mylist.py*

```
class MyList:
    def __init__(self, start):
        #self.wrapped = start[:]           # Copy start: no side effects
        self.wrapped = list(start)       # Make sure it's a list here
    def __add__(self, other):
        return MyList(self.wrapped + other)
    def __mul__(self, time):
        return MyList(self.wrapped * time)
    def __getitem__(self, offset):
        # Also passed a slice on [:]
        return self.wrapped[offset]      # For iteration if no __iter__
    def __len__(self):
        return len(self.wrapped)        # Also fallback for truth tests
    def append(self, node):
        self.wrapped.append(node)
    def __getattr__(self, name):
        # Other methods: sort/reverse/etc.
        return getattr(self.wrapped, name)
    def __repr__(self):
        # Catchall display method
        return repr(self.wrapped)

if __name__ == '__main__':
    x = MyList('hack')
    print(x)
    print(x[2])
    print(x[1:])
    print(x + ['code'])
    print(x * 3)
    x.append('1'); x.extend(['z'])
    x.sort()
    print(' '.join(c for c in x))

$ python3 mylist.py
['h', 'a', 'c', 'k']
c
['a', 'c', 'k']
['h', 'a', 'c', 'k', 'code']
['h', 'a', 'c', 'k', 'h', 'a', 'c', 'k', 'h', 'a', 'c', 'k']
1 a c h k z
```

Note that it's also important to copy the start value by calling `list` instead of slicing here, because otherwise the result may not be a true *list*, and so will not respond to expected list methods, such as `append` (e.g., slicing a string returns another string, not a list). You would be able to copy a `MyList` start value by slicing because its class overloads the slicing operation and provides the expected list interface; however, you need to avoid slice-based copying for objects such as strings.

3. *Subclassing*: One solution appears in [Example B-25](#); your solution will be similar. You can also use `super` here instead of explicit superclass names for methods and attributes, as partly noted in the code's comments:

Example B-25. Part6/mysub.py

```
from mylist import MyList

class MyListSub(MyList):
    calls = 0                                # Shared by instances
    def __init__(self, start):
        self.adds = 0                        # Varies in each instance
        MyList.__init__(self, start)        # Or: super().__init__(start)

    def __add__(self, other):
        print('add: ' + str(other))
        MyListSub.calls += 1                # Class-wide counter
        self.adds += 1                      # Per-instance counts
        return MyList.__add__(self, other)  # Or: super().__add__(other)

    def stats(self):
        return self.calls, self.adds        # All adds, my adds

if __name__ == '__main__':
    x = MyListSub('read')
    y = MyListSub('code')
    print(x[2])
    print(x[1:])
    print(x + ['lp6e'])
    print(x + ['book'])
    print(y + ['py312'])
    print(x.stats())
```

\$ python3 mysub.py

```
a
['e', 'a', 'd']
add: ['lp6e']
['r', 'e', 'a', 'd', 'lp6e']
add: ['book']
['r', 'e', 'a', 'd', 'book']
add: ['py312']
['c', 'o', 'd', 'e', 'py312']
(3, 2)
```

4. *Attribute methods*: The following works through this exercise. As noted in [Chapter 28](#) and elsewhere, `__getattr__` is *not* called for built-in operations in Python 3.X, so the expressions aren't intercepted at all here; a class like this must somehow redefine `__X__` operator-overloading methods explicitly. You can find more on this limitation in [Chapters 28, 31, 32, and 38](#), as well as *workarounds* for it in [Chapter 39](#) and its *inheritance* special case in [Chapter 40](#). Its impacts are potentially broad but can be addressed with code:

```
$ python3
>>> class Attrs:
    def __getattr__(self, name):
        print('get:', name)
    def __setattr__(self, name, value):
        print('set:', name, value)

>>> x = Attrs()
>>> x.append
get append
>>> x.lang = 'py312'
set: lang py312
>>> x + 2
```

```

TypeError: unsupported operand type(s) for +: 'Attrs' and 'int'
>>> x[1]
TypeError: 'Attrs' object is not subscriptable
>>> x[1:5]
TypeError: 'Attrs' object is not subscriptable

```

5. *Set objects*: Here's the sort of interaction you should get. To make the import of *Chapter32/setwrapper.py* work, either run this in the folder where this file resides, copy this file to your working directory, or add this file's folder to your import search path per [Part V](#). Comments explain which methods are called. Also, bear in mind that sets are a built-in type in Python, so this is mostly just a coding exercise (see [Chapter 5](#) for more on sets):

```

$ python3
>>> from setwrapper import Set      # Run there, copy here, or mod path
>>> x = Set([1, 2, 3, 4])          # Runs __init__
>>> y = Set([3, 4, 5])

>>> x & y                          # __and__, intersect, then __repr__
Set:[3, 4]
>>> x | y                          # __or__, union, then __repr__
Set:[1, 2, 3, 4, 5]

>>> z = Set('hello')              # __init__ removes duplicates
>>> z[0], z[-1], z[2:]            # __getitem__
('h', 'o', ['l', 'o'])

>>> for c in z: print(c, end=' ')  # __iter__ (else __getitem__)
...
h e l o
>>> ''.join(c.upper() for c in z) # __iter__ (else __getitem__)
'HELO'

>>> len(z), z                     # __len__, __repr__
(4, Set:['h', 'e', 'l', 'o'])

>>> z & 'mello', z | 'mello'
(Set:['e', 'l', 'o'], Set:['h', 'e', 'l', 'o', 'm'])

```

A solution to the multiple-operand extension subclass looks like the class in [Example B-26](#). It needs to replace only two methods in the original set. The class's documentation string explains how it works:

Example B-26. *Part6/multiset.py*

```

from setwrapper import Set

class MultiSet(Set):
    """
    Inherits all Set names, but extends intersect and union to support
    multiple operands. Note that "self" is still the first argument
    (stored in the *args argument now). Also note that the inherited
    & and | operators call the new methods here with 2 arguments, but
    processing more than 2 requires a method call, not an expression.
    intersect doesn't remove duplicates here: the Set constructor does.
    """
    def intersect(self, *others):
        res = []
        for x in self:
            for other in others:
                if x not in other: break
            else:
                res.append(x)
        return Set(res)

```

```

def union(*args):
    res = []
    for seq in args:
        for x in seq:
            if not x in res:
                res.append(x)
    return Set(res)

```

self is args[0]
For all args
For all nodes
Add new items to result

Your interaction with this extension will look something like the following. Note that you can intersect by using `&` or calling `intersect`, but you must call `intersect` for three or more operands; `&` is a binary (two-sided) operator. Also, note that we could have called `MultiSet` simply `Set` to make this change more transparent if we used `setwrapper.Set` to refer to the original within `multiset` (the `as` clause in an import could rename the class too if desired):

```

>>> from multiset import *
>>> x = MultiSet([1, 2, 3, 4])
>>> y = MultiSet([3, 4, 5])
>>> z = MultiSet([0, 1, 2])

>>> x & y, x | y
(Set:[3, 4], Set:[1, 2, 3, 4, 5])

>>> x.intersect(y, z)
Set:[]

>>> x.union(y, z)
Set:[1, 2, 3, 4, 5, 0]

>>> x.intersect([1,2,3], [2,3,4], [1,2,3])
Set:[2, 3]

>>> x.union(range(10))
Set:[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]

>>> w = MultiSet('soap')
>>> w
Set(['s', 'o', 'a', 'p'])

>>> ''.join(w | 'super')
'soapuer'

>>> (w | 'super') & MultiSet('slots')
Set(['s', 'o'])

```

Two operands
Three operands
Four operands
Non-MultiSets work, too
String sets

6. *Class tree links*: [Example B-27](#) lists one way to change the `lister` class in [Example 31-10](#), along with a rerun of the associated tester to show its augmented format. For full credit, do the same for the `dir`-based version, and also do this when formatting class objects in the tree-climber variant.

To import `testmixin.py` as a test, either copy it over from the Chapter 31 examples folder or add that folder to `sys.path` as we did earlier in [Part IV](#)'s solutions. It was copied here for variety:

Example B-27. `Part6/listinstance-mod.py`

```

class ListInstance:
    def __attrnames(self):
        ...unchanged...

    def __str__(self):
        return (f'<Instance of {self.__class__.__name__}'
                f'({self.__supers()}), '
                f'address {id(self):#x}: '
                f'{self.__attrnames()}>')

```

My class's name
My class's supers
My address (hex)
name=value list

```

def __supers(self):
    names = []
    for super in self.__class__.__bases__:
        names.append(super.__name__)
    return ', '.join(names)

    # Or: ', '.join(super.__name__ for super in self.__class__.__bases__)

if __name__ == '__main__':
    import testmixin
    testmixin.tester(ListInstance)

$ python3 listinstance-mod.py
<Instance of Sub(Super, ListInstance), address 0x10edc66c0:
  data1='code'
  data2='Python'
  data3=3.12
>

```

7. *Composition*: A full-points solution is coded in [Example B-28](#), with comments from the description mixed in with the code. This is one case where it's probably easier to express a problem in code than it is in narrative:

Example B-28. Part6/lunch.py

```

class Lunch:
    def __init__(self):
        self.cust = Customer()
        self.empl = Employee()
    def order(self, foodName):
        self.cust.placeOrder(foodName, self.empl)
    def result(self):
        self.cust.printFood()

class Customer:
    def __init__(self):
        self.food = None
    def placeOrder(self, foodName, employee):
        self.food = employee.takeOrder(foodName)
    def printFood(self):
        print(self.food.name)

class Employee:
    def takeOrder(self, foodName):
        return Food(foodName)

class Food:
    def __init__(self, name):
        self.name = name

if __name__ == '__main__':
    x = Lunch()
    x.order('burritos')
    x.result()
    x.order('pizza')
    x.result()

```

When run, customers place orders and get food from employees. This could be much more involved, but it suffices to demo the routing of messages between objects that's typical in OOP code:

```
$ python3 lunch.py
burritos
pizza
```

8. *Zoo animal hierarchy*: **Example B-29** shows one way to code the taxonomy in Python; it's artificial, but the general coding pattern applies to many real structures, from GUIs to employee databases to spacecraft. Notice that the `self.speak` call in `Animal` triggers an independent inheritance search, which generally finds `speak` in a subclass. Test this interactively by calling the `reply` method for instances per the exercise description. Try extending this hierarchy with new classes and making instances of various classes in the tree:

Example B-29. Part6/zoo.py

```
class Animal:
    def reply(self):    self.speak()           # Back to subclass
    def speak(self):   print('blah')         # Custom message

class Mammal(Animal):
    def speak(self):   print('huh?')

class Cat(Mammal):
    def speak(self):   print('meow')

class Dog(Mammal):
    def speak(self):   print('bark')

class Primate(Mammal):
    def speak(self):   print('Hello world!')

class Hacker(Primate): pass                 # Inherit from Primate
```

Part VII, Exceptions

See “**Test Your Knowledge: Part VII Exercises**” on page 945 in **Chapter 36** for the exercises.

1. `try/except`: One possible coding of the `oops` function is listed in **Example B-30**. As for the non-coding questions, changing `oops` to raise a `KeyError` instead of an `IndexError` means that the `try` handler won't catch the exception—it “percolates” to the top level and triggers Python's default error message. The names `KeyError` and `IndexError` come from the outermost built-in names scope (the *B* in “LEGB”). Import builtins and pass it as an argument to the `dir` function to see this for yourself, per **Chapter 17**.

Example B-30. Part7/oops.py

```
def oops():
    raise IndexError()

def doomed():
    try:
        oops()
    except IndexError:
```

```

        print('caught an index error!')
    else:
        print('no error caught...')

if __name__ == '__main__': doomed()

```

```

$ python3 oops.py
caught an index error!

```

2. *Exception objects and lists:* [Example B-31](#) is one way to extend this module for an exception of its own:

Example B-31. Part7/oops2.py

```

class MyError(Exception): pass

def oops():
    raise MyError('Hack!')

def doomed():
    try:
        oops()
    except IndexError:
        print('caught an index error!')
    except MyError as exc:
        print('caught error:', MyError, exc)
    else:
        print('no error caught...')

if __name__ == '__main__':
    doomed()

$ python3 oops2.py
caught error: <class '__main__.MyError'> Hack!

```

Like all class exceptions, the raised instance is accessible via the `as` variable `data`; the error message shows both the class's (`<...>`) and its instance's (`Hack!`) displays. The instance must be inheriting both an `__init__` and a `__repr__` or `__str__` from Python's `Exception` class, or it would print much as the class does. See [Chapter 35](#) for details on how these defaults work in built-in exception classes.

3. *Error handling:* [Example B-32](#) is one way to solve this exercise. It codes tests in a file rather than interactively, but the results are similar enough for full credit. Notice that the empty `except` and `sys.exc_info` approach used here will catch exit-related exceptions that listing `Exception` with an `as` variable won't; that's probably not ideal in most applications code but might be useful in a tool like this designed to work as a sort of exceptions firewall.

Example B-32. Part7/exctools.py

```

import sys, traceback

def safe(callee, *pargs, **kargs):
    try:
        callee(*pargs, **kargs)
    except:
        traceback.print_exc()
        print(f'Got {sys.exc_info()[0]} {sys.exc_info()[1]}')

```

```
if __name__ == '__main__':
    import oops2
    safe(oops2.oops)
```

\$ **python3 exctools.py**

Traceback (most recent call last):

```
File ".../LP6E/AppendixB/Part7/exctools.py", line 5, in safe
    callee(*pargs, **kargs)          # Catch everything else
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File ".../LP6E/AppendixB/Part7/oops2.py", line 4, in oops
    raise MyError('Hack!')
oops2.MyError: Hack!
Got <class 'oops2.MyError'> Hack!
```

Bonus points: the sort of code in [Example B-33](#) could turn this into a *function decorator* that could wrap and catch exceptions raised by any function, using techniques introduced briefly in [Chapter 32](#), but covered more fully in [Chapter 39](#)—it augments a function, rather than expecting it to be passed in explicitly, and produces similar output when run (there’s an extra call level, and file-names differ):

Example B-33. Part7/exctools_deco.py

```
import sys, traceback

def safe(callee):
    def callproxy(*pargs, **kargs):
        try:
            return callee(*pargs, **kargs)
        except Exception as E:
            traceback.print_exc()
            print(f'Got {E.__class__} {E}')
    return callproxy

if __name__ == '__main__':
    import oops2

    @safe
    def test():
        oops2.oops()

    test()
```

4. *Self-study examples:* In closing, Examples [B-34](#) through [B-43](#) are 10 examples for you to study on your own. Their code and supporting files are in the *Self-Study-Demos* subfolder of the examples package’s *AppendixB/Part7* folder. These require no extra installs as they use standard-library tools, though tkinter is sketchy on phones (see [Appendix A](#)). For more examples, see follow-up books and resources for the application domains you’ll be exploring next:

Example B-34. Part7/Self-Study-Demos/largest_dir.py

Find the largest Python source file in a single directory

```
import os, glob
dirname = '/Users/me/Downloads'    # Edit me to use (or use input() or sys.argv)

allsizes = []
```



```

allpy = glob.glob(dirname + os.sep + '*.py')
for filename in allpy:
    filesize = os.path.getsize(filename)
    allsizes.append((filesize, filename))

allsizes.sort()
print(allsizes[:2])
print(allsizes[-2:])

```

Example B-35. Part7/Self-Study-Demos/largest-tree.py

Find the largest Python source file in an entire directory tree

```

import sys, os, pprint
if sys.platform[:3] == 'win':
    dirname = r'C:\Users\me\Downloads' # Edit me to use
else:
    dirname = '/Users/me/Downloads'

allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirname):
    for filename in filesHere:
        if filename.endswith('.py'):
            fullname = os.path.join(thisDir, filename)
            fullsize = os.path.getsize(fullname)
            allsizes.append((fullsize, fullname))

allsizes.sort()
pprint.pprint(allsizes[:2])
pprint.pprint(allsizes[-2:])

```

Example B-36. Part7/Self-Study-Demos/largest-import.py

Find the largest Python source file on the module import search path

```

import sys, os, pprint
visited = {}
allsizes = []
for srcdir in sys.path:
    for (thisDir, subsHere, filesHere) in os.walk(srcdir):
        thisDir = os.path.normpath(thisDir)
        if thisDir.upper() in visited:
            continue
        else:
            visited[thisDir.upper()] = True
        for filename in filesHere:
            if filename.endswith('.py'):
                pypath = os.path.join(thisDir, filename)
                try:
                    pysize = os.path.getsize(pypath)
                except:
                    print('skipping', pypath)
                allsizes.append((pysize, pypath))

allsizes.sort()
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])

```

Example B-37. Part7/Self-Study-Demos/summer1.py

Sum columns in a text file separated by commas

```
filename = 'data.txt'    # Edit me for others
sums = {}

for line in open(filename):
    cols = line.split(',')
    nums = [int(col) for col in cols]
    for (ix, num) in enumerate(nums):
        sums[ix] = sums.get(ix, 0) + num

for key in sorted(sums):
    print(key, '=', sums[key])
```

Example B-38. Part7/Self-Study-Demos/summer2.py

Similar to summer1, but using lists instead of dictionaries for sums

```
import sys
filename = sys.argv[1]          # "python3 summer2.py data.txt 3"
numcols = int(sys.argv[2])
totals = [0] * numcols

for line in open(filename):
    cols = line.split(',')
    nums = [int(x) for x in cols]
    totals = [(x + y) for (x, y) in zip(totals, nums)]

print(totals)
```

Example B-39. Part7/Self-Study-Demos/regptest.py

Simple test for regressions in the output of a set of scripts

```
import os
testscripts = [dict(script='test1.py', args=''),      # Edit me to use (or glob)
               dict(script='test2.py', args='-opt')]  # Add encodings if needed

for testcase in testscripts:
    commandline = '%(script)s %(args)s' % testcase
    output = os.popen(commandline).read()
    result = testcase['script'] + '.result'
    if not os.path.exists(result):
        open(result, 'w').write(output)
        print('Created:', result)
    else:
        priorresult = open(result).read()
        if output != priorresult:
            print('FAILED:', testcase['script'])
            print(output)
        else:
            print('Passed:', testcase['script'])
```

Example B-40. Part7/Self-Study-Demos/gui1.py

```
"""
Build a GUI with tkinter having buttons that change color and grow.
Caution: this GUI may grow until you close its window manually!
"""

from tkinter import *
import random
fontsize = 25
colors = ['red', 'green', 'blue', 'yellow', 'orange', 'white', 'cyan', 'purple']

def reply(text):
    print(text)
    popup = Toplevel()
    color = random.choice(colors)
    Label(popup, text='Popup', bg='black', fg=color).pack()
    L.config(fg=color)

def cycle():
    L.config(fg=random.choice(colors))
    win.after(250, cycle)

def grow():
    global fontsize
    fontsize += 5
    L.config(font=('arial', fontsize, 'italic'))
    win.after(100, grow)

win = Tk()
L = Label(win, text='Hack',
          font=('arial', fontsize, 'italic'), fg='yellow', bg='navy',
          relief=RAISED)
L.pack(side=TOP, expand=YES, fill=BOTH)
Button(win, text='popup', command=(lambda: reply('new'))).pack(side=BOTTOM, fill=X)
Button(win, text='cycle', command=cycle).pack(side=BOTTOM, fill=X)
Button(win, text='grow', command=grow).pack(side=BOTTOM, fill=X)
win.mainloop()
```

Example B-41. Part7/Self-Study-Demos/gui2.py

```
"""
Similar to gui1, but use classes so each window has own state info.
Caution: this GUI may grow until you press Stop or kill its window!
"""

from tkinter import *
import random

class MyGui:
    """
    A GUI with buttons that change color and make the label grow
    """
    colors = ['blue', 'green', 'orange', 'red', 'brown', 'yellow']

    def __init__(self, parent, title='popup'):
        parent.title(title)
        self.growing = False
        self.fontsize = 10
        self.lab = Label(parent, text='Hack2', fg='white', bg='navy')
```

```

        self.lab.pack(expand=YES, fill=BOTH)
        Button(parent, text='Hack', command=self.reply).pack(side=LEFT)
        Button(parent, text='Grow', command=self.grow).pack(side=LEFT)
        Button(parent, text='Stop', command=self.stop).pack(side=LEFT)

    def reply(self):
        "change the button's color at random on Hack presses"
        self.fontsize += 5
        color = random.choice(self.colors)
        self.lab.config(bg=color,
                        font=('courier', self.fontsize, 'bold italic'))

    def grow(self):
        "start making the label grow on Grow presses"
        self.growing = True
        self.grower()

    def grower(self):
        "multiple presses schedule multiple growers"
        if self.growing:
            self.fontsize += 5
            self.lab.config(font=('courier', self.fontsize, 'bold'))
            self.lab.after(500, self.grower)

    def stop(self):
        "stop all button grower loops on Stop presses"
        self.growing = False

class MySubGui(MyGui):
    colors = ['black', 'purple']          # Customize to change color choices

MyGui(Tk(), 'main')
MyGui(Toplevel())
MySubGui(Toplevel())
mainloop()

```

Example B-42. Part7/Self-Study-Demos/popmail.py

```

"""
POP email inbox scanning and deletion utility.
Scan pop email box, fetching just headers, allowing
deletions without downloading the complete message.
"""

import poplib, getpass, sys

mailserver = 'your pop email server name here' # Edit me: your pop.server.net
mailuser   = 'your pop email user name here'   # Edit me: your userid
mailpasswd = getpass.getpass(f'Password for {mailserver}? ')

print('Connecting...')
server = poplib.POP3(mailserver)
server.user(mailuser)
server.pass_(mailpasswd)

try:
    print(server.getwelcome())
    msgCount, mboxSize = server.stat()
    print('There are', msgCount, 'mail messages, size ', mboxSize)
    msginfo = server.list()

```

```

print(msginfo)
for i in range(msgCount):
    msgnum = i+1
    msgsize = msginfo[1][i].split()[1]
    resp, hdrlines, octets = server.top(msgnum, 0)           # Get hdrs only
    print('-'*80)
    print('%d: octets=%d, size=%s' % (msgnum, octets, msgsize))
    for line in hdrlines: print(line)

    if input('Print?') in ['y', 'Y']:
        for line in server.retr(msgnum)[1]: print(line)     # Get whole msg
    if input('Delete?') in ['y', 'Y']:
        print('deleting')
        server.dele(msgnum)                                # Delete on srvr
    else:
        print('skipping')

finally:
    server.quit()                                           # Make sure we unlock mbox
    input('Bye.')                                           # Keep window up on Windows

```

Example B-43. Part7/Self-Study-Demos/sqlbase.py

Database script to populate and query an SQLite database, stored in people.db

```

import sqlite3, time
conn = sqlite3.connect('people.db')    # Filename for database storage
curs = conn.cursor()                  # Submit SQL through cursor

# Make+fill table if doesn't yet exist
tbl = curs.execute('select name from sqlite_master where name = \'people\'')
if tbl.fetchone() is None:
    print('Making table anew')
    curs.execute('create table people (name, job, pay)')

    recs = [('Pat', 'mgr', 40000), ('Sue', 'dev', 60000), ('Bob', 'dev', 50000)]
    for rec in recs:
        curs.execute('insert into people values (?, ?, ?)', rec)
    conn.commit()

# Show all rows
print('Rows:')
curs.execute('select * from people')
for row in curs.fetchall():
    print(row)

# Show just devs
print('Devs:')
curs.execute("select name, pay from people where job = 'dev'")
colnames = [desc[0] for desc in curs.description]
while row := curs.fetchone():
    print('-' * 30)
    for (name, value) in zip(colnames, row):
        print(f'{name:<4} => {value}')

# Update devs' pay: shown on next run
secs = int(time.time()) # UTC!
curs.execute('update people set pay = ? where job = ?', [secs, 'dev'])
conn.commit()

```


Symbols

- # comments, 358
- \$ character in interactive coding, 30
- * (asterisk) use in code, 270

A

- absolute imports, 604, 605
- abstract superclasses, 722-724, 784
 - library tools, 723
- access-by-key files, 224
- aggregation
 - composition, 782
 - object, 697
- AI (artificial intelligence), 10
- Ajax (Asynchronous JavaScript and XML), 9
- aliasing, 424
- __all__ variable, 619-620
- Android
 - interactive coding and, 28
 - Python installation, 27
 - Python on, 1148-1151
- annotations, functions, 383, 468-470
- anonymous functions, 384-385
- AOT (ahead-of-time) compilers, 40
 - PyThran, 23
 - Shed Skin, 23
- arbitrary arguments, argument matching, 433-436
- arbitrary expressions, 58
- arbitrary scope nesting, 409
- architecture, 560
 - (see also program architecture)
- argument matching, 428-429
 - arbitrary arguments, 433-436
 - defaults, 429
 - function calls, 429
 - unpacking arguments, 434-435
 - function definitions, 429
 - collecting arguments, 433-434
 - generalized set functions example, 446-448
 - keyword-only arguments, 437-439
 - keywords, 429, 431-433, 452
 - minimum value calculation, 443-446
 - mins.py example, 443-446
 - passing arguments, 430-431
 - positional-only arguments, 439-440
 - print function example, 448-451
 - syntax, 429-430, 471
- argument ordering
 - function calls, 442
 - boundary cases, 442
 - formal definitions, 442
 - function definitions, 440
 - boundary cases, 441
 - formal definitions, 441
- argument passing, 423
 - defaults, 431-433
 - function calls, 430
 - function definitions, 430
 - immutable arguments, 424
 - keywords, 431-433
 - matching arguments, 430-431
 - multiple results simulation, 427
 - mutable arguments, 424
 - changes, 426-427
 - output parameter simulation, 427
 - references, shared, 424-426
- arguments, 383, 1168
 - arbitrary, 1168
 - decorators, 1042, 1055-1057
 - callable, 1042
 - class decorators, 1067
 - descriptors, 998-999
 - immutable, 424

- keyword-only, 449
- mutable, 424
 - coupling, 456
- passing, 382
- passing functions as, 465
- positional, 1076-1078
- scopes, 416-419
- super function, 857-858
- validation, 1075-1076
- ArithmeticError exception class, 917
- arrays, associative, 185
- artificial intelligence (AI), 10
- as clause, 625-626
- ASCII, 950-951
 - Latin-1 and, 952
 - UTF-8 and, 952
- aspect-oriented programming, 1034
- assert statement
 - as conditional raise statement, 902
 - constraint trapping, 903
- assignments
 - argument passing, 423-427
 - augmented, 263, 272-274
 - basic assignment, 263-264
 - class attributes, 714
 - extended-unpacking, 263, 266-268
 - boundary cases, 268-269
 - for loops, 270
 - function calls, 271
 - function headers, 270
 - implicit, 262
 - from, 579
 - import, 579
 - lists, 262
 - module attribute creation, 582
 - multiple-targets, 263, 271
 - shared references, 271-272
 - mutables, 372
 - name references, nested scopes, 406
 - named assignment expression, 263, 274-277
 - names, 261, 395, 396
 - namespaces, 726-728
 - object references, 261
 - scopes, 398
 - sequence assignments, 264-266
 - sequences, 262
 - __setitem__ method, 744
 - statements, 247, 262
 - syntax, 270
 - tuples, 262
- associative arrays, 185
- async function, 517
 - concurrent tasks
 - async def, 519-520
 - async for, 523-524
 - async with, 523-524
 - as_completed, 522-523
 - await, 519-520
 - gather, 522-523
 - I/O (input/output) operations, 517
 - multitasking, 518
 - serial tasks, 518
 - tasks to avoid, 520-522
- async statement, 382
- attribute accessors
 - __get__ method, 992
 - __getattr__ method, 992
 - __getattribute__ method, 992
 - __set__ method, 992
 - __setattr__ method, 992
- attribute fetches, 146, 853
 - computing value, 995
 - __getattr__ method, 1007-1012
 - __getattribute__ method, 992, 1007-1012
 - inheritance, 672
 - loop avoidance, 1009
 - properties, 993
- attribute interception, 991
 - descriptors, 837
- attribute management, 991
 - access, 992-993
 - built-in attributes, intercepting, 1017-1020
 - computed attributes, 995
 - descriptors, 997, 1015
 - computed attributes, 1002
 - method arguments, 998-999
 - properties and, 1006-1007
 - read-only, 999-1000
 - state information, 1003-1006
 - validation and, 1023-1026
 - __getattr__ method, 1007-1016
 - validation, 1027-1028
 - __getattribute__ method, 1007-1016
 - validation, 1028-1029
 - properties, 993, 1015
 - decorators, 996-997
 - validation and, 1021-1023
 - validation
 - descriptors and, 1023-1026
 - properties and, 1021-1023
- attribute trees, 718
- attributes, 561-563
 - assignment, 760-761, 1115
 - assignments, 992

- changing, 860-861
- class statement, 714
 - assignment-rule exception, 716
 - defs, 715
 - visibility, 715
- classes, 701-702
- computing, 1012-1014
- conflict resolution, 801-802
- deleting, 760-761
- descriptors, 837, 1002
- functions, 383
- __getattr__ method, 759
- __getattribute__ method, 836
- instance objects, 680
- instances, 701-702
- introspection tools, 699-700
- listing, 358
 - name strings, 358
 - type names, 359
- listing, mix-in classes, 802
 - class trees, 807-811
 - __dict__, 803-806
 - __dir__, 806-807
 - __main__, 622-625
- metaclasses, 1108
- modules, 575
 - classes, 666-667
 - creating, 582
 - names, qualification, 584-585
- mutable, 861-862
- __name__, 622-625
- names, namespaces, 725-726
- nested modules, 596
- private, class decorators, 1064-1067
- properties, 834
 - assignment support, 835
- pseudoprivate, 788-791
- public, class decorators, 1067-1075
- reference interception, 759
- __self__, 789
- __setattr__ method, 760-761
- slots, 672, 826
- __slots__, 829
- user-defined, 467
- augmented assignments, 263, 272-274
- await statement, 382

B

- backslash characters, 375, 1166
 - strings, 133-134
- backslash escape sequences, 63
 - f-string literal, 164

- BaseException exception class, 916
- __bases__ attribute, 672, 699, 731, 797, 823, 1112
- basic assignment, 263-264
- Beautiful Soup, 9
- BeeWare Toga, 8
- benchmarking, 529
 - iteration, 535-538
 - function calls, 536-538
 - map function, 536-538
 - timeit module, 540-549
 - timing module
 - example, 530-533
 - test functions, 533-534
- binary data, 949, 954
 - struct module, 978
- binary files, 217-218, 955-956
- binary-mode files, 972, 973
 - bytes object, 974
- blank lines, 372
- block strings, 137-138
- blocks, statements, 300-302
- BOM (byte order marker), 953
 - adding, 985-987
 - text editors, 984-985
 - text-mode files, 972
- __bool__ method, 775
- Boolean type, 232
- Boolean values, 775
- Booleans, 78, 308-309
- Boost.Python, 9
- bound methods, 791, 838
 - callback functions, 773
- bounds, 66
- break statements, 313
 - interactive loops, 255
- breaks, 49, 1159
- buffering
 - files, 215
 - output, 214
- built-in docstrings, 362-363
- built-in exception classes, 916
 - ArithmeticError, 917
 - BaseException, 916
 - Exception, 917
 - LookupError, 917
 - OSError, 917
- built-in functions
 - handling by assignment, 553
 - super, 691
- built-in modules, module search path and, 567
- built-in objects, 54-55
 - collections, 54

- dictionaries, 54
- lists, 54
- search tables, 54
- built-in operations
 - delegating, 698, 1070
 - operator overloading methods, 1071-1075
- built-in tools, 12, 941
- built-in types, 817
 - class creation, 822
 - classes, 821
 - cyclic data structures, 237
 - extending
 - embedding, 818-819
 - subclassing, 819-821
 - immutable types, 237
 - references, 235
 - repetition, 236
- built-ins, 399
 - generators, 503-504
 - LEGB rule, 400-401
- byte files, 76-77
- byte strings, 966
 - bytearray object, 970-972
 - bytes constructor, 968
 - formatting, 968
 - methods, 966
 - mixing types, 969
 - sequence operations, 967-968
- bytearray object, 955, 970-972
- bytecode, 6, 19, 21, 26, 564
 - executing, 567
 - files, 19
 - hash-based files, 565
 - import optimization, 20
 - interactive prompt code, 20
 - magic number, 565
 - PyPy, 7
 - Python versions, 19
 - startup-speed optimization, 19
 - timestamp-based files, 565
- bytes object, 949, 954
 - binary-mode files, 974

C

- call expressions, methods, 146
 - operator overloading, 770-773
- __call__ method, 770-773
- call proxies, 1032
- callback functions
 - bound methods, 773
 - lambda functions, 773
- callbacks, 772-773

- bound methods, 794
- calls
 - def statements, 388
 - functions, 386
- CFFI, 9
- chaining exceptions, 900-902
- Chaquopy, 9
- character code conversion, 144
- Cinder, 24
- circular imports (see recursion)
- circular references, garbage collection, 119
- __class__ attribute, 672, 699, 800, 821, 852, 1111
- class decorators, 845, 848-850, 1031, 1038, 1104
 - arguments, 1067
 - built-in types, 1062
 - class managers, 1032
 - implementation, 1038-1039
 - instances, multiple, 1039-1040, 1062-1064
 - interface proxies, 1032
 - metaclasses comparison, 1092-1093
 - object interfaces, 1059
 - private attributes, 1064-1067
 - public attributes, 1067-1075
 - singleton classes, 1057-1062
 - state, enclosing scopes and, 1067
- class managers, 1032
- class statement, 80, 649, 713
 - attributes, 714
 - assignment-rule exception, 716
 - defs, 715
 - visibility, 715
 - syntax, 713-714
- class trees
 - attribute listings, 807-811
 - inheritance, 1113
 - OOP, 654-656, 1186
- class-based exceptions, 940
- classes, 649
 - attributes, 701-702
 - changing, 860-861
 - descriptors, 1001
 - introspection tools, 699-700
 - mutable, 861-862
 - pseudoprivate, 788-791
 - slots, 672
 - __bases__ attribute, 672, 699, 731, 797, 823, 1112
 - built-in operation override, 668
 - built-in types, 821
 - __class__ attribute, 672, 699, 800, 821, 852, 1111
 - composites, 695-698
 - composition, 650
 - conflation, 604

- delegation-based, 864
- descriptors, 997
- versus dictionaries, 673-675
- docstrings, 361, 734-735
- exception classes, 911
- extending, 650
- generators, 503-504
- inheritance, 649, 650, 718, 823
 - attribute trees, 718
 - inherited methods, 719-720
 - metaclass subclasses and, 823
 - MRO (method resolution order), 855
 - multiple inheritance, 796-814
 - privacy, 719
 - subverting, 689
- instances, 650
 - attributes, 1045
 - creating, 822
- metaclasses (see metaclasses)
- method objects, 791-794
- methods, 683, 837, 841
 - adding, 1103-1105
 - double underscores, 668
 - encapsulation, 684
 - instance counting, 842-844
 - metaclass comparison, 1119
 - method calls, 716-717
- module attributes, 666-667
- __mro__ attribute, 800, 811, 823, 854, 1111
- namespaces
 - hierarchies, 650
 - inheritance, 714
- nested
 - descriptors, 1001
 - scope, 728-730
- object factories, 795-796
- objects, 661
 - default behavior, 662
- OOP, 652, 679
 - constructors, 680-681
 - inheritance, 664-666
 - intercepting operators, 667-670
 - methods, 683-686
 - methods, augmenting, 689-691
 - namespaces, 653
 - operator overloading, 686-687
 - polymorphism, 691-692
 - subclassing, 688-690
 - superclasses, 652, 665
 - testing, 681-682
- persistence, 786
- polymorphism, 657-658
 - as program units, 650
 - redundancy, 649, 684
 - scopes, 863-864
 - state retention, 416
 - storage, 864
 - subclasses, 665
 - superclass constructors, 864
 - types and, 821
 - user-defined, 821-822
 - versus modules, 735
- classmethod function, 840
- closure functions, 772
- closures, 407-409
- code folders
 - dedicated, 29
 - directory structure, 30
 - per-chapter subfolders, 29
- code obfuscation, list comprehensions and, 489
- code points, 133
 - Unicode, 951
 - escapes, 962
 - value, 960
- code reuse, 870
 - exceptions, 911
 - functions, 379
 - modules, 560
 - OOP, 656-659
- code strings, module imports, 629
- code wrapping, 865
- coding gotchas, 372-373
- collections, 54
 - argument matching, 428
- collector modules, 1180
- colons
 - compound statement headers, 372
 - statements, 248
- columns in code, 250
- command line, 47
 - launchers, 45
 - program file running, 35-36
 - program files, 34
 - shell programs, 36
 - tools, 8
- commas
 - expressions, 239
 - tuples, 209-210
- comments, 32, 35
 - interactive coding, 30
- comparison operators, 773
- comparisons, 228-231
 - core object types, 229
 - dictionaries, 230

- lists, 176
- mixed-type, 230
- recursive, 228
- relative magnitude, 229
- compiled extensions, 7
- compilers
 - AOT (ahead-of-time), 40
 - Python-to-Wasm, 40
- completion certificate, 1128-1131
- complex numbers, 56, 81, 224
- component integration, 4, 9
- composites
 - delegation, 696
 - nesting, 695
- composition
 - abstract superclasses, 784
 - aggregation, 782
 - has-a relationships, 782-786
 - OOP, 782-786
 - stream processors, 784
- compound objects, 228
- compound statements, 72, 248, 291
 - if statements, 291
 - multiple-choice selection, 293-295
 - match statements, 295
 - attribute patterns, 298
 - if statements, 296
 - instance patterns, 298
 - literal patterns, 297
 - mapping patterns, 297
 - nested patterns, 298
 - nesting, 296
 - parentheses, 298
 - sequence patterns, 297
 - syntax, 299
 - blocks, 300-302
 - delimiters, 302
 - indentation, 300-302
 - special syntax, 303-304
- comprehensions, 1172
 - converting tuples, 210
 - dictionaries, 191-192
 - dictionary, 504, 1174
 - for loops, 327-328
 - generators and, 504
 - list comprehensions, 67, 347, 483
 - files, 348-349
 - if clause, 349
 - nested for loops, 350
 - sequence operations, 210
 - syntax, 68
 - lists, 182-183
 - loop variables, 397
 - set, 504
 - syntax, 483
 - type calls, 504
 - variables, scopes and, 504-505
- concatenation, 130
 - list literals, 183
 - strings, 132, 138
- concept hierarchy, 245
 - expressions, 245
 - statements, 245
- constraints
 - assert statement, 903
 - MicroPython, 24
- constructor convolution, 740
- constructors
 - classes, 680-681
 - customizing, 693-694
 - __init__ method, 656
 - operator overload, 740
 - superclass methods, 720
 - superclasses, 694, 864
- container object length, 238
- __contains__ method, 756-758
- context managers, 223
 - with statement
 - context-management protocol, 905-907
 - multiple context managers, 907-908
- continuation lines in interactive coding, 30
- continue statements, 313
- control language, 6
- control-nesting, exception handlers, 928-930
- conversions
 - text files, 218-220
 - tuples, 210-211
- cooperative method dispatch, 855-857
- copies
 - build-in types, 235
 - of objects, 226-228
- copy method, 71
- core object types, 55
- coroutines, 382
- counter loops, range object, 324-325
- coupling, 455
 - variables, 456
- coupling modules, 617
- CPython, 22-24, 26
 - WebAssembly/Emscripten platform, 40
- cross-file changes, 404-405
- CSV module, object storage, 222
- current working directory (CWD), 214
- customization by inheritance, 688

- CWD (current working directory), 214
- cycles, 49, 1159
 - recursion, 463
- cyclic data, 1159
 - structures, 237
- Cython, 7, 9, 10, 23, 26

D

- data descriptors, 998
 - inheritance, 1116
- data hiding, 603, 618, 788
 - __all__ variable, 619-620
 - __dir__ function, 620-621
 - __getattr__ function, 620-621
 - __X prefix, 619-620
- databases, 9
 - API, 9
 - books database example, 194-195
 - object storage, 704
 - pickle module, 704
 - shelf object updates, 708
 - shelve database, 705-707
 - shelve module, 704, 705
- debugging, 943, 1158
 - exceptions and, 49
 - exercise, 48-50
 - try statement, 934-935
- decimals, 56, 81, 168
- declaration statements, 115
- decoding, Unicode, 952
- decoration, 1031
 - functions, 468-470
- decorator timers, 540
- decorators
 - arguments, 1042, 1055-1057
 - callable, 1042
 - function annotations, 1085
 - call tracers, 1043-1044
 - callables returning callables, 1035
 - class decorators, 845, 848-850, 1031, 1038, 1104
 - arguments, 1067
 - built-in types, 1062
 - class managers, 1032
 - implementation, 1038-1039
 - instances, multiple, 1039-1040, 1062-1064
 - interface proxies, 1032
 - metaclasses comparison, 1092-1093
 - object interfaces, 1059
 - private attributes, 1064-1067
 - public attributes, 1067-1075
 - singleton classes, 1057-1062
 - state, enclosing scopes and, 1067
 - class management, 1042
 - class methods, 1105-1108
 - function decorators, 845-847, 1031
 - argument validation, 1075-1087
 - call proxies, 1032
 - function call timing, 1053-1055
 - function managers, 1032
 - metafunctions, 1034
 - runtime declarations, 1034
 - syntax, 1034
 - usage, 1034
 - user-defined, 847-848
 - function management, 1042
 - function validation
 - keywords, 1078-1081
 - positional arguments, 1076-1078
 - functions, 383
 - implementation, 1035-1036
 - versus macros, 1034
 - methods, 1037, 1048
 - descriptors, 1051-1053
 - nested functions, 1050
 - nesting, 1040-1041, 1085
 - properties, 996
 - deleters, 996
 - setters, 996
 - range-testing, 1076-1087
 - state retention
 - class-instance attributes, 1045
 - enclosing-scope nonlocals, 1046-1047
 - function attributes, 1047-1048
 - global variables and, 1046
 - syntax, 1033
 - timer decorator, 1057
 - wrapper function objects, 1041
- def statement, 381, 383
 - function calls, 388
 - name assignments, 396
 - nonlocal statement and, 409-412
 - runtime execution, 384
- default argument values, scopes, 416-419
- definition, functions, 385
- __del__ method, 776-777
- delegation, 696
 - versus inheritance, 1066
 - like-a relationships, 786-788
 - OOP, 786-788
- delimiter string, 149
- deprecation policies, 278
- descriptors, 837, 997, 1015
 - arguments, 998-999
 - attributes, computed, 1002

- class attributes, 1001
- classes, 997
- data descriptors, 998
 - inheritance, 1113
- decorating methods, 1051-1053
- files, os module, 224
- hiding, 998
- inheritance, 1001
- inheritance and, 998
- nested classes, 1001
- operator overloading, 1073-1075
- properties and, 1006-1007
- read-only, 999-1000
- `__set__` method, 998
- slots, 1007
- state, 1003-1006
- design
 - cross-file changes, 404-405
 - functions, 379
 - global variable minimization, 402-403
 - modules, 617
 - patterns, factory functions, 407-409
- destructors, 776
- developer productivity, 3, 5
- development implications, 21
- development tools, 941
 - debuggers, 943
 - documentation tools, 942
 - error-checking tools, 942
 - IDEs, 942
 - installation management, 944
 - optimization, 943
 - profilers, 943
 - shipping, 943
 - testing tools, 942
- DFLR (depth first, then left to right), 797
- diagonals, 487
- diamonds, user-defined, 799
- `__dict__` attribute, 582, 583-584, 626, 627-629
- dictionaries, 54, 56, 69, 81, 168
 - backward indexing, 195
 - books database example, 194-195
 - building, 189-191
 - characteristics, 184-186
 - classes versus dictionaries, 673-675
 - collections, 185
 - comparisons, 230
 - comprehensions, 191-192, 504, 1174
 - syntax, 485
 - converting to tuples, 212
 - copying, 71
 - empty, 186
 - function annotations, 468
 - in operator, 187
 - indexing, 69, 240, 1162
 - initializing, 191
 - iteration, 73-74, 342
 - key insertion order, 192-193
 - key order, 185
 - keys, 184
 - immutable objects, 196
 - integer keys, 196
 - mapping to, 194-195
 - missing, 197
 - sorting, 201-202
 - storage, 187
 - tuple keys, 197
 - keys method, 187
 - len function, 187
 - lists, sorting, 180
 - literal expressions, 186
 - literal unpacking, 190
 - literals, 185
 - magnitude comparisons, 202
 - methods, 71, 188-189
 - module namespaces, 582-584
 - mutability, 69, 185, 187-188
 - namespaces, 671, 730-732
 - slots, 826, 827-828
 - nesting, 70-71, 198
 - new indexes, 196
 - object references, 185
 - operations, 185
 - mapping, 69-70
 - sequence operations, 196
 - sets, 200-201
 - sorting, 375, 1166
 - union operator, 193-194
 - updating, 71
 - views, 200-201
 - views objects, 199-200
 - zip object, 330
- dictionary keys, 240, 1162
 - if statements, 71-73
 - indexing and, 69
 - insertion order, 70
- dictionary tools, 1170
- dictionary-based formatting expressions, 156-157
- dir function, 62
 - object attributes, 358
 - name strings, 358
 - type names, 359
- direct recursion, 459
- directories

- commands, 36
- module search path, 568
 - .pth path-file, 569
 - PYTHONPATH, 568, 570
 - site-packages, 569
 - standard-library directories, 569
- directory paths, 217
 - filename, 214
- disabling code, triple-quoted strings, 138
- Django, 8
- do until loop, 312
- docstrings, 138, 357, 360, 734-735
 - (see also documentation)
 - built-in, 362-363
 - content, 361
 - indentation, 362
 - literals, 360
 - object classes, 361
 - print, 361
 - raw strings, 360
 - standards, 362
 - triple-quoted block, 360
 - user-defined, 360-362
- documentation, 49, 357
 - (see also docstrings; Pydoc)
 - # comments, 358
 - dir function, 358-359
 - priorities, 362
 - Sphinx, 370
- documentation tools, 942
- dotted-path syntax, packages, 594
 - module import, 596
 - relative imports, 605
- Durus, 9
- dynamic typing, 12
 - declaration statements, 115
 - objects, 117
 - polymorphism and, 124
 - variables, 116-117

E

- ease of use, 13
- else clause, 316-317
- else statements, 313
- embedding
 - built-in object types, 818-819
 - embedded objects, 696
- empty data structures, 231
- empty dictionaries, 186
- empty strings, 216
- encapsulation, 684, 779, 870
 - packaging, 779
- enclosing scopes, 1036
 - functions, 552
 - state retention and, 1067
- encoding
 - source files, 964-966
 - Unicode, 952
 - UTF-8, 952
- end-of-line character, 284
- enumerate function, loops, 331-332
- equality, testing for, 229
- error handling, 1189
 - exceptions, 876
 - testing user input, 256-257
 - try statement, 257-258
- error messages, 937
- error-checking tools, 942
- errors, 1158
 - exercise, 48
- escape characters, 132-135
 - raw strings, 136-137
- eval function, 144
- event notification, 876
- Excel, 10, 45
- exception classes, 911
 - built-in, 916
 - ArithmeticError, 917
 - BaseException, 916
 - Exception, 917
 - LookupError, 917
 - OSError, 917
 - categories, catching, 913
 - Exception, 917
 - exception details, 913
 - hierarchies, 912, 914-916
 - raising instances, 913
 - state and, 911
 - superclasses, 913
- exception groups, 923-925
 - nesting, 929
- exception handlers, 48
 - nesting, 927
 - control-flow, 928-930
 - syntactic nesting, 930-931
 - sys.exc_info, 935-937
 - try/except/else, 933-934
- exception superclass, 913
- exceptions, 875
 - as string objects, 898
 - assert statement, 902-903
 - (see also assert statement)
 - built-in
 - catching, 891

- custom printing, 919-920
- default printing, 918-919
- state, 918-919, 920-922
- catching, 878-879
- categories, 917
- chaining, 900-902
- class instance objects, 911
- class-based, 940
- code reuse, 911
- connection closure, 934
- control flow, 876
- debugging and, 49
- default behaviors, 890
- details, 921
- error checks, 884
- error handling, 876
- error messages, 937
- event notification, 876
- except clause, 938-940
- exception handler, 877-878
- file closing, 934
- hierarchies, 912, 914-916
- methods, 921-922
- no exception case, 889
- propagating, 900
- raise statement, 898-902, 932
 - (see also raise statement)
- raising, 879-880, 898
- scopes, 899-900
- as signals, 932-933
- special-case handling, 876
- superclasses, 912
- termination, 876, 881-882
- termination handlers, 908-909
- tests, in-process, 935
- traceback object, 937
- try statement, 885-897
 - (see also try statement)
- debugging and, 934-935
- tuples, 887
- unexpected, 889
- user-defined, 880
 - nonerror conditions, 933
- variables, scopes, 397
- with statement, 904-909
 - (see also with statement)
- wrappers, 938

exec function, 44-45

executable statements

- from, 579
- import, 579

executables, standalone, 24, 1152-1155

execution

- deferring, 470
- speed, 6

exponentiation, 56

expression statements, 280-281

- in-place changes, 281

expressions, 32, 114

- arbitrary, 58, 484
- commas, 239
- concept hierarchy, 245
- curly braces, 56
- dictionaries, 56
- function-related, 379
- generator expressions, 483, 490, 500, 803
 - as iterables, 497
 - best uses, 497
 - versus filter function, 499
 - generator objects, 496
 - versus map function, 497-499
 - slicing sequences, 508
- if/then/else, 306-307
- indexing expressions, 58
 - slicing, 58
- lambda, 381
- list comprehensions, 484
 - mapping, 484
- lists, 56
- literal expressions, 56
- method call expressions, 146
- named assignment expression, 263, 274-277
- objects, 53
- operator overload, 740
- slicing expression, 142
- square brackets, 56
- string formatting, 152-154
 - advanced, 155-156
 - dictionary-based, 156-157
 - type codes, 154
- strings, 130

extended slicing, 142

extended-unpacking assignments, 266-268

- boundary cases, 268-269
- for loops, 270, 321-322

extension modules, 576

extensions, 941

F

f-strings, 284

- literals, 162
 - backslash, 164
 - custom formats, 164
 - quotes, 164

- REPL, 163
 - substitutions, 163
- factorials, 1176
- factory functions, 407-409, 795-796
 - metaclasses, 1100
- false values, 231-233
- FIFO (first in, first out), queues, 224
- file extensions, imports, 373
- file icons, program running, 37
- file objects
 - creating, 213
 - print operations, 282
- file operations, 213
- file-processing, 216-217
- filenames, 980-983
 - directory path, 214
 - modules, 575-576
 - program files, 18
- files, 75, 81
 - access-by-key, 224
 - binary, 217-218
 - binary-mode, 972
 - buffered, 215
 - byte files, 76-77
 - bytecode, 19
 - close method, 215
 - content, 215
 - context managers, 223
 - cross-file changes, minimizing, 404-405
 - descriptor files, os module, 224
 - directory paths, 217
 - FIFOs, 224
 - iteration, 76
 - iterators, 215, 216
 - list comprehensions, 348-349
 - methods export, 224
 - newline characters, 216
 - number types, 224
 - object storage, 218-220
 - open function, 77
 - opening, 214
 - pipes, 224
 - program architecture, 561
 - running as standalone script, 622-625
 - sets, 224
 - shell-command streams, 224
 - shelves, 224
 - sockets, 224
 - standard streams, 223
 - string types, 224
 - strings, 215
 - empty, 216
 - text files, 215
 - conversions, 218-220
 - pathname, 973
 - text output, 75
 - text-mode, 972
 - Unicode and, 76-77
- filter function, 477-478, 499
- finalization (see termination actions)
- find method, 149
- first-class object model, 465-466
- Flask, 8, 9
- floating point numbers, 56, 81, 114, 168
 - conversion, 144
- files, 224
 - interactive loops, 258-259
- folder hierarchies, package imports, 594
- folders
 - packages, 593
 - names, 595
 - nesting, 595
- for loops, 73-74, 318
 - comprehensions, 327-328
 - data types, 319
 - dictionary comprehensions, 191
 - extended-unpacking assignments, 270, 321-322
 - in functions, 387-388
 - __getitem__ method, 745
 - iteration protocol, 491
 - lists, 327-328
 - nested, 322-323
 - list comprehensions, 350, 486
 - sequence assignment, 319-321
 - sequence scans, 325-326
 - targets, 318
 - tips, 372
 - tuple assignment, 319-321
- FOSS (free and open source software), 7
- fractions, 81, 168
 - files, 224
- frameworks, 659
- free and open source software (FOSS), 7
- from * statement, 577-578
 - __all__ variable, 619-620
 - __init__.py files, 603
 - variable meanings, 640
 - _X prefix, 619-620
- from statement, 577
 - as clause, 625-626
 - interactive statement, 641
 - link problems, 639
 - potential problems, 580-582
 - reloads, 640

- frozen binaries, 24
- function calls, 31
 - argument matching, 428, 429
 - unpacking arguments, 434-435
 - argument ordering, 442
 - boundary cases, 442
 - formal definitions, 442
 - argument passing, 430
 - arguments, mutable, 424
 - assignments, 271
 - benchmarks, 536-538
 - changeable data, 415
 - interfaces, 772-773
 - parentheses, 373
 - timing, 1053-1055
 - tracing, decorators, 1043-1044
- function decorators, 845-847, 1031
 - argument validation, 1075-1076
 - call proxies, 1032
 - function call timing, 1053-1055
 - function managers, 1032
 - metafunctions, 1034
 - runtime declarations, 1034
 - syntax, 836, 1034
 - usage, 1034
 - user-defined, 847-848
- function definitions
 - argument matching, 429, 433-434
 - argument ordering, 440
 - boundary cases, 441
 - formal definitions, 441
 - argument passing, 430
- function headers, 270
- function managers, 1032
- function validation
 - keywords, 1078-1081
 - positional arguments, 1076-1078
- functional programming, 3, 475, 662
 - closures, 407-409
 - first-class object model, 465
 - map function, 476-477
- functions
 - annotations, 125, 383, 468-470
 - anonymous, 384-385, 470
 - (see also lambda expression)
 - argument matching syntax, 429
 - arguments, 383
 - coupling, 455
 - passing, 382
 - validation, 1075-1087
 - async, 517-525
 - async statement, 382
 - attributes, 383
 - state, 1047-1048
 - state retention, 414-415
 - user-defined, 467
 - await statement, 382
 - built-in, handling by assignment, 553
 - calls, 386
 - def statement, 388
 - classmethod, 840
 - closure functions, 772
 - cohesion, 456
 - coroutines, 382
 - coupling, 455
 - decorations, 468-470
 - decorators, 383
 - def statements, 381, 383, 384
 - defaults, 550-552
 - deferring execution, 470
 - definitions, 382, 385
 - dir, 62
 - eval, 144
 - exec, 44-45
 - execution environment, 456
 - factory functions, 1100
 - filter, 477-478
 - first-class object model, 465-466
 - for loops in, 387-388
 - generator functions, 483, 490, 491, 500
 - best uses, 493
 - iteration protocol, 492
 - send method, 494
 - slicing sequences, 507-508
 - state suspension, 490-491
 - yield statement, 494-495
 - generators, 382
 - global statement, 382
 - global variables, 455
 - helper functions, 1090-1092
 - inlining, 470
 - intersect, 388
 - lambda body expression, 388
 - polymorphism, 389
 - introspection, 466, 1082
 - lambda expressions, 381, 384-385
 - len, 133
 - loop variables, 552
 - manager functions, 1090
 - metafunctions, 845
 - names, locals, 549-550
 - nested, decorating methods, 1050
 - nonlocal statement, 382
 - objects, mutable, 550-552

- open, 77
- overview, 379
- parameters, 383
- passing as arguments, 465
- plain functions, 791, 838
- recursive (see recursion)
- reduce, 478-479
- reload, 587-590
- return statements, 381, 383
- returns, 552
- scopes, enclosing, 552
- set, 77
- size, 456
- slicing sequences, 507
- staticmethod, 840
- str, 57
- super function, 851
 - argument lists, 857-858
 - attribute-fetch algorithm, 853
 - call-chain anchors, 855-857
 - deployment, 854
 - MRO algorithm, 852-853
 - noncalls, 858-859
 - operator overloading, 858-859
- testdriver, 935
- timer utility functions, 531
- type, 78-79
- type hints, 125
- uses, 380-381
- yield statement, 382
- __future__ import, 622

G

- game programming, 10
- garbage collection, 71
 - circular references, 119
 - objects, 118-119
- generalized set functions, argument matching, 446-448
- generator expressions, 483, 490, 500, 803
 - as iterables, 497
 - best uses, 497
 - versus filter function, 499
- generator objects, 496
- versus map function, 497-499
- nesting, 498
- slicing sequences, 508
- generator functions, 483, 490, 491, 500
 - best uses, 493
 - iterable objects, 749
 - iteration protocol, 491, 492
 - send method, 494

- slicing sequences, 507-508
- state suspension, 490-491
- yield statement, 494-495
- generator objects, 500, 752-756
 - generator expressions, 496
- generators, 382
 - built-ins, 503-504
 - classes, 503-504
 - comprehensions, 504
 - memory footprint, 511
 - permutation, 511-513
 - results generation, 505-506
 - single-pass iterables, 501-503
 - __get__ method, 992
 - __getattr__ method, 695, 759, 864, 992, 1007-1012
 - versus __getattr__ method, 1014-1015
 - __getattr__ method, 836, 992, 1007-1012, 1016
 - versus __getattr__ method, 1014-1015
 - __getitem__ method, 742
 - iteration, 745-746
 - iteration and, 745
 - slice expressions, 743-744
- global namespaces, 725
- global scope, state retention and, 413
- global statement, 382, 401-402
 - emulation, 405-406
- global variables
 - coupling, 455
 - cross-file changes, 404-405
 - multithreading, 403
 - program design, 402-403
 - state, decorators and, 1046
- glyphs, 954
- graphics processing, 10
- __gt__ method, 773-774
- GUIs (graphical user interfaces), 7, 8
 - IDLE, 38-39

H

- has-a relationships, composition, 782-786
- hash-based bytecode files, 565
- hashes, 185
- Haskell, 483
- helper functions, 1090-1092
- hex escapes, 960
- hierarchies, subclassing, 693
- HPy, 9

I

- I/O (input/output) operations, async extension, 517
- __iadd__ method, 770

- IDEs (integrated development environments), 39, 942
 - attribute lists, 359
 - IDLE, 38
- IDLE, 1139-1141
- IDLE GUI, 38-39
- if else expression, 72
 - recursion, 458-459
- if statements, 248, 291
 - dictionary keys, 71-73
 - list comprehensions, 349
 - match statements, 296
 - multiple-choice selection, 293
 - function definition, 294
 - switch defaults, 294
 - syntax, 72
- if/then/else expression, 306-307
- image processing, 10
- immutability, 82
 - strings, 59-60, 168
 - tuples, 74, 210-211
- immutable arguments, 424
- immutable sequence of 8-bit integers, bytes type, 954
- immutable sequence of characters, str type, 954
- immutable sequences, strings, 130
- immutable types, 169, 241, 1164
- implementation alternatives, 22-24
- implementation-related objects, 55
- import models
 - modules, 609
 - namespace packages, 609-613
 - module searches, 610-611
 - package-relative imports, 604-609
 - packages, 593-594, 609
 - module searches, 594-595
 - package creation, 595-600
 - package initialization, 603-604
- import optimization, bytecode, 20
- import statement, 576-577
 - as clause, 625-626
 - modules, 564
- imports
 - as runtime operations, 564
 - circular, 1181
 - __future__, 622
 - modules, 561-564
 - bytecode, 564-566
 - bytecode execution, 567
 - initialization code, 578
 - namespace nesting, 586-587
 - one-time, 578
 - previous imports, 565
 - runtime assignment, 579-580
 - statement order, 638-639
 - variables, 585-586
 - nested, 1179
 - nested modules, attributes, 596
 - package-relative, 593
 - absolute, 604, 605
 - name clashes, 638
 - relative, 604-609
 - packages, 593, 594, 609, 1180
 - folder hierarchies, 594
 - from statement, 596
 - recursive, 642-643
- in-place changes, 121-123
 - expression statements, 281
 - names, 395
- incremental prototyping, 681
- indentation
 - blocks, 249-252
 - consistency, 372
 - text editors, 252
- index assignments, 177-178
- __index__ method, 745
- indexing, 240, 1161
 - dictionaries, 240, 1162
 - expressions, 58
 - slicing, 58
- instances
 - iteration, __getitem__, 745-746
 - operator overloading, 742-745
- lists, 176-177
- out of bounds, 240
- string indexing, 241, 1163
- strings, 140-143
- indirect recursion, 459
- infinite loops, 312
- inheritance, 779, 1181
 - attribute fetches, 672
 - built-ins, 1116
 - class trees, 1113
 - classes, 649, 718
 - attribute trees, 718
 - customization, 664-666
 - inheritance search, 823
 - metaclass subclasses and, 823
 - privacy, 719
 - composites, 695-698
 - customization by, 688
 - data descriptors, 1113, 1116
 - versus delegation, 1066
 - descriptors, 998, 1001

- inherited methods, 719-720
- is-a relationships, 780-782
- metaclasses, 1089, 1108, 1109, 1111-1112
 - inheritance relationships, 1108
 - names acquisition, 1110
- MRO (method resolution order), 797, 799-800, 823
- multiple, 796
 - attribute conflicts, 801-802
 - DFLR, 797
 - inheritance sources, 811-814
 - mix-in classes, 802-811
 - order, 862-863
 - overview, 797-799
 - slots, 831, 833
- namespaces, 714
- nonclass instances, 823
- OOP, 650, 780-782
 - attributes, 651-653
 - class customization, 664-666
 - multiple inheritance, 654
- operator-overloading methods, 741
- properties, 994
- secondary trees, 824
- single-inheritance, slots, 832
- string representation methods, 763
- subclassing, 692
- subverting, 689
- super function, 1116
- inheritance algorithm, 1112-1118
- inheritance hierarchy, 649
- `__init__` method
 - constructors, 656
 - operator overload and, 740
- `__init__.py` files, 597-598, 603
 - from `*` statement, 603
 - module namespace initialization, 603
 - package initialization, 603
 - searches, 603
- inlining functions, 470
- inputs, user (see user inputs)
- installation
 - Android, 27
 - Linux, 27
 - macOS, 27
 - Unix, 27
 - Windows, 27
- installation management, 944
- instance methods, 838, 840
 - metaclass comparison, 1120-1122
- instance objects, 661, 662
 - attributes, 680
- instances
 - attribute privacy, 761-762
 - attributes, 701-702
 - class decorators, 1039-1040, 1062-1064
 - classes, 650
 - counting, 841-842
 - class methods, 842-844
 - creation, 822
 - indexing
 - iteration, 745-746
 - operator overloading, 742-745
 - literal syntax, 821
 - metaclasses, 822
 - nonclass instances, 822
 - inheritance, 823
 - OOP, 652
 - namespaces, 653
 - state, 1003
 - state, decorators and, 1045
 - subclasses, namespace dictionaries, 731
- integer keys, 196
- integers, 56, 81, 168
 - files, 224
 - `__index__` method, 745
 - precision in large numbers, 56
- interaction, 1157
- interactive coding
 - `$` character, 30
 - Android, 28
 - command line, 47
 - comments, 30
 - exercise, 48
 - macOS, 28
 - prompt, 32-34
 - prompts, 30
 - REPLs (real-eval-print loops)
 - code folders, 29-30
 - IPython, 30
 - starting, 28-29, 47
 - running code, 31-32
 - Windows, 28
- interactive command-line
 - REPLs (real-eval-print loops), 28
- interactive loops, 254-255
 - error handling
 - input testing, 256-257
 - try statements, 257-258
 - floating-point numbers, 258-259
 - math on user inputs, 255-256
 - nesting code, 259
- interactive tests, 241
 - from statement, 641

- reload statement, 641
- interface proxies, wrappers, 1032
- internet modules, 8
- interpreted language, 6
- interpreter, 17-20, 26
- intersect function, 388
 - lambda body expression, 388
 - local variables, 389-390
 - polymorphism, 389
- introspection
 - __dict__ attribute, 626
 - listing modules, 627-629
 - functions, 1082
- introspection tools, 698
 - class attributes, 699-702
 - display tools, 700-701
 - instance attributes, 701-702
 - tool classes, 702
- iOS, Python on, 1151-1152
- IPython, 30
- IronPython, 9, 22
- is-a relationships, inheritance, 780-782
- items method, 73
- __iter__ method, 746
 - user-defined iterables, 747-749
 - yield function and, 752-756
- iterable objects, 336, 746
 - combining items, 478-479
 - generator functions, 749
 - item selection, 477-478
 - mapping functions over, 476-477
- iterable, definition, 336
- iterables
 - generator expressions as, 497
 - generators, 501-503
 - map function, 514
 - single-pass, 501-503
 - user-defined
 - __iter__ method, 747-749
 - yield function, 752-756
 - zip function, 515
- iteration, 76, 336, 1172
 - benchmarking, 535-538
 - function calls, 536-538
 - map function, 536-538
 - dictionaries, 73-74, 342
 - filter built-in, 344-345
 - generators, 483
 - __getitem__ method, 745
 - instance indexing, __getitem__, 745-746
 - list comprehensions, 348, 486
 - lists, 182
 - map built-in, 344-345
 - multiple iterations, 749-751
 - multiple-pass iterables, 345-346
 - Python morph and, 526
 - single-pass iterables, 345-346
 - single-scan, 748
 - standard-library iterables, 346
 - strings, 139
 - text files, 215
 - view objects, 199
- iteration protocol, 335-339, 746
 - for loops, 491
 - full iteration protocol, 339-341
 - generator functions, 491, 492
 - iter, 339, 341-342, 492
 - iteration tools, 338
 - lists, 343
 - manual iteration, 341
 - next, 339, 341
 - range object, 342
 - readlines, 338
 - StopIteration exception, 337
 - zip object, 343
- iteration tools, 211, 351-355
- iterator objects, multiple iterations, 749-751
- iterators
 - definition, 336
 - nesting, 344

J

- Java, 9
- JIT compilers
 - Numba, 23
 - PyPy, 23
- join method, 149
- json module, 978-979
- JSON, object storage, 222
- Jupyter notebooks, 10
- Jupyter Notebooks, 40
- JVM (Java Virtual Machine)
 - Jython, 22
- Jython, 9, 22

K

- keys
 - dictionaries, 184
 - insertion order, 192-193
 - integer keys, 196
 - mapping values to, 194-195
 - missing, 197
 - sorting, 201-202

- tuple keys, 197
- keys method, 73
- keyword arguments
 - list order, 179
 - print operations, 284
- keyword-only arguments, 437-439
- keywords, 1169-1170
 - argument matching, 428, 429, 452
 - argument passing, 431-433
 - function validation, 1078-1081
 - required, 438
- Kivy, 8, 1153

L

- lambda expression, 381, 384-385, 471
 - argument-matching syntax, 471
 - callbacks, 480
 - code proximity and, 473
 - loops in, 474
 - name assignments, 396
 - names, 394
 - nesting, 475
 - nesting logic, 474
 - reasons to use, 472-473
 - selection logic, 474
- lambda functions
 - callbacks, 773
 - wrapper layers, 1041
- Latin-1, 952
- learning curve, 13
- LEGB rule, 396-398, 728-730
 - built-ins, 400-401
- len function, 133, 238
 - sequence reordering, 326
- lexical scoping, 394, 586
- libraries, 4
 - (see also standard library)
 - utilities, 12
- like-a relationships, 786-788
- lines comparison, text files, 908
- Linux
 - Python installation, 27
 - Python on, 1146-1148
- list comprehension pattern, map function, 514
- list comprehensions, 67, 347, 483
 - best use, 489-490
 - code obfuscation and, 489
 - diagonals, 487
 - expression mapping, 484
 - expressions, 484
 - arbitrary, 484
 - files, 348-349
 - filter function, 499
 - for clauses, nested, 486
 - if clause, 349, 485
 - iteration, 486
 - map function, 484
 - matrixes and, 487-490
 - nested for loops, 350
 - sequence operations, 210
 - syntax, formal, 485
- lists, 54, 81, 168
 - as literal expression, 175
 - assignments, 262
 - bounds, 66
 - breadth-first traversal, 461
 - building, 203
 - characteristics, 173-175
 - compared to tuples, 211
 - comparisons, 176, 229
 - comprehensions, 182-183
 - deleting one item, 180
 - __dict__ attribute, 627-629
 - dictionaries, sorting, 180
 - index assignments, 177-178
 - indexing, 176-177
 - iteration, 182
 - iteration protocol, 343
 - literals, 174
 - unpacking, 183
 - maps, 182-183
 - method calls, 178-179
 - mutability, 174
 - nesting, 66-67
 - operations, 174
 - operators, 175
 - ordering, 179-180
 - reversing, 180
 - searching, 170
 - sequence operations, 65
 - slice assignments, 177-178, 184
 - slicing, 176-177
 - type-specific operations, 65-66
- literal expressions, 56
 - dictionaries, 186
 - lists as, 175
- literal patterns, 297
- literals, 55
 - dictionaries, 185
 - docstrings, 360
 - f-string formatting, 162
 - backslash, 164
 - custom formats, 164
 - quotes, 164

- REPL, 163
 - substitutions, 163
- lists, unpacking, 183
- string formatting, 152
- local scopes, 394
- local variables, 389-390
 - recursion, 395
 - static locals, 467
- logic alternatives, 375
- LookupError exception class, 917
- loops, 1165
 - attribute interception methods, 1009
 - coding, 374
 - comprehensions, 397
 - counters, range object, 324-325
 - defaults, 418-419
 - dictionary comprehensions, 191
 - else clause, 316-317
 - enumerate function, 331-332
 - for loops, 73-74, 318, 372
 - data types, 319
 - extended-unpacking assignments, 321-322
 - in functions, 387-388
 - __getitem__, 745
 - nested, 322-323, 350
 - sequence assignment, 319-321
 - targets, 318
 - tuple assignment, 319-321
- infinite, 312
- interactive, 254-255
 - floating-point numbers, 258-259
 - math on user inputs, 255-256
 - nesting code, 259
 - testing inputs, 256-257
 - try statements, 257-258
- lambda expression, 474
- nested, multiple, 931-932
- object indexes, 312
- offsets, 331-332
- versus recursion, 459-460
- sequence reordering, 326
- sequence scans, 325-326
- skipping items, 327
- strings and, 139
- variables, 552
- while loops, 311-312
 - break statement, 313, 315-316
 - continue statement, 313-315
 - do until, 312
 - else statement, 313
 - pass statement, 313-314
- zip object, 328-330

- __lt__ method, 773-774

M

- machine code, 26
- macOS
 - interactive coding and, 28
 - Python installation, 27
 - Python on, 1141-1146
- magnitude comparisons, 305
- __main__ attribute, 622-625
- __main__.py files, 599-600
- manager functions, 1090
- mangled names, 788
- manual iteration, 341
- manuals, standard, 370-371
- map function, 476-477
 - benchmarking, 536-538
 - emulating, 513-517
 - iterables, multiple, 514
 - list comprehension pattern, 514
 - list comprehensions, 484
 - nesting, 498
 - versus generator expressions, 497-499
- mapping, 82
 - mutability, 185
 - operations, 69-70
- mapping patterns, 297
- mappings, 168
- match statement, 72
- match statements, 295
 - attribute patterns, 298
 - if statements, 296
 - instance patterns, 298
 - literal patterns, 297
 - mapping patterns, 297
 - nested patterns, 298
 - nesting, 296
 - parentheses, 298
 - sequence patterns, 297
 - syntax, 299
 - blocks, 300-302
 - delimiters, 302
 - indentation, 300-302
 - special syntax, 303-304
- math module, 57
- mathematical operations, 56
 - user inputs, 255-256
- matplotlib, 10
- matrixes, list comprehensions, 487-490
- memory management, 12
 - generators, 511
 - object memory, 71

- metaclass/class dichotomy, 824-825
- metaclasses, 822, 1097
 - attributes, 1108
 - class creation, 1089
 - class decorators comparison, 1092-1093
 - class statements, 1104
 - types, 1095-1097
 - construction, 1099
 - decorating class methods, 1105-1108
 - factory functions, 1100
 - inheritance, 823, 1089, 1099, 1108, 1109, 1111-1112
 - names acquisition, 1110
 - inheritance relationships, 1108
 - method protocol, 1097
 - methods, 1089, 1118
 - adding, 1103-1105
 - class methods comparison, 1119
 - instance methods comparison, 1120-1122
 - operator overloading in, 1120
 - nonclass instances, 1101-1103
 - superclasses comparison, 1110-1111
 - types, 1093-1094
 - subclasses, 1094
 - when to use, 1090
- metafunctions, 845
 - function decorators, 1034
- metaprograms, 626
- method calls, 716-717
 - lists, 178-179
 - syntax, 146
- method calls (OOP), 653-654
- method objects, 791-794
 - bound methods, 791
 - plain functions, 791
- methods
 - attribute fetches, 146
 - `__bool__`, 775
 - bound methods, 838
 - byte strings, 966
 - call expressions, 146
 - `__call__`, 770-773
 - class methods, 838, 840, 841
 - classes, 837
 - adding to, 1103-1105
 - method calls, 716-717
 - `__contains__`, 756-758
 - decorators, 1037, 1048
 - descriptors, 1051-1053
 - nested functions, 1050
 - `__del__`, 776-777
 - dictionaries, 188-189
 - double underscores, 668
 - encapsulation, 684
 - find, 149
 - `__get__`, 992
 - `__getattr__`, 695, 759, 992
 - `__getattribute__`, 992
 - `__getitem__`, 743-744
 - `__gt__`, 773-774
 - `__iadd__`, 770
 - `__index__`, 745
 - inherited, 719-720
 - instance, 838
 - instance methods, 840
 - `__iter__`, 746, 747-749
 - join, 149
 - `__lt__`, 773-774
 - metaclasses, 1089, 1097, 1118-1122
 - `__next__`, 746
 - OOP, 655
 - operator overloading, 740-742
 - plain-function, 838-839
 - `__radd__`, 766
 - replace, 148, 149
 - `__repr__`, 763
 - scopes, 863-864
 - `__set__`, 992
 - `__setattr__`, 760-761, 992
 - split, 150
 - standard-library manual, 147
 - static, 717, 837-840
 - `__str__`, 763-765
 - string formatting, 152
 - format method, 157-162
 - string methods, 146
 - syntax, 146
 - strings, 60-62
 - subclasses, 689-690
 - superclass, 719
 - tuples, 210-211
 - `__setitem__`, 744
- methods (OOP), 663
- MicroPython, 24
- Microsoft Store, 1136
- minimalism of Python, 4
- mins.py, 443-446
- mix-in classes
 - attribute lists, 802
 - class trees, 807-811
 - `__dict__`, 803-806
 - `__dir__`, 806-807
- mixed-type comparisons, 230
- module packages (see packages)

- module search path, 567
 - built-in modules, 567
 - configuring, 570
 - file selection, 571-572
 - file sources, 572
 - home directory, 568
 - packages, 572, 594
 - .pth path-file directory, 569
 - PYTHONPATH directory, 568-570
 - site-packages directory, 569
 - standalones, 572
 - standard-library directories, 569
 - sys.path list, 570-571
- modules, 34, 1158
 - attributes, 42-44, 575
 - classes, 666-667
 - creating, 582
 - __dict__, 583-584
 - names, qualification, 584-585
 - classtools, 702
 - code reuse, 560
 - cohesion, 617
 - collector, 1180
 - coupling, 617
 - creating, 575-576
 - defining, 575
 - designing, 617
 - exercise, 48
 - extension modules, 576
 - file names, 42
 - filenames, 575-576
 - files, 34
 - from * statement, 577-578
 - from statement, 559, 577
 - potential problems, 580-582
 - global scope, 560
 - import statement, 559, 576-577
 - importing, 41, 47, 561-564
 - bytecode, 564-566
 - bytecode execution, 567
 - previous imports, 565
 - importlib.reload, 559
 - imports
 - direct calls, 630
 - initialization code, 578
 - name strings, 629-638
 - namespace nesting, 586-587
 - one-time, 578
 - runtime assignment, 579-580
 - statement order, 638-639
 - variables, 585-586
 - json, 978-979
 - listing, __dict__ attribute, 627-629
 - __main__, 622-625
 - math module, 57
 - mutables, changing, 579
 - __name__, 622-625
 - namespaces, 560
 - dictionaries, 582-584
 - generating, 582-583
 - initialization, 603
 - nesting, 586-587
 - partitioning, 560
 - nested, attributes, 596
 - pickle, 704, 978-979
 - program architecture, 41, 561
 - random, 57
 - re, 977
 - redundancy, 560
 - reload function, 587-590
 - reloads, 42, 630
 - recursive reloaders, 631-638
 - scopes
 - imports and, 583
 - variables, 585-586
 - shelve, 704, 705
 - standard-library modules, 563
 - statements, 53
 - import, 582
 - strings, 130
 - struct, 978
 - using, 576-582
 - versus classes, 735
- MRO (method resolution order), 797, 799-800
 - attribute fetch, 853
 - class inheritance, 855
 - class trees, 1113, 1115
 - deployment, 854
 - as flattened tree, 823
 - inheritance, 823
 - super function, 852-853
- __mro__ attribute, 800, 811, 823, 854, 1111
- multiline statements, 32
- multiline strings, 137-138
- multiline text, triple-quoted strings, 138
- multiple inheritance, 796
 - attribute conflicts, 801-802
 - DFLR (depth first, left to right), 797
 - inheritance sources, 811-814
 - mix-in classes, attribute lists, 802-811
 - MRO (method resolution order), 797, 799-800
 - order, 862-863
 - overview, 797-799
 - slots, 831

- multiple-target assignments, 263, 271
 - shared references, 271-272
- multiplication, 56
- multitasking, async function, 518
- multithreading, global variables, 403
- multiway branching, dictionaries, 473
- mutability
 - dictionaries, 69, 185, 187-188
 - in-place changes, 122
 - lists, 174
 - mutable defaults, 433
 - objects, changing in place, 224
 - strings, 59-60
- mutable arguments, 424
 - changes, 426-427
 - coupling, 456
- mutable objects
 - changing, 579
 - functions, 550-552
- mutable types, 169
- mutables, assignments, 372

N

- name annotations, basic assignment, 264
- name assignments, nested scopes, 406
- __name__ attribute, 622-625
- name clashes, 581, 625, 638
- name collisions, tool classes, 702
- name mangling, 788
- name references, nested scopes, 406
- name strings, module imports, 629
 - code strings, running, 629
 - direct calls, 630
 - reloads, 630-638
- named assignment expression, 263, 274-277
- named tuples, 212-213
- names
 - aliasing, 424
 - assignments, 261, 395, 396
 - def statement, 394
 - def statements, 396
 - global statements, 396
 - in-place changes, 395
 - lambda expressions, 394
 - lambda statements, 396
 - LEGB rule, 396-398
 - local, 395
 - nonlocal statements, 396
 - references, 396
 - variables, 576
- namespace packages, 593, 598, 609
 - module searches, 610-611
- namespaces, 393, 560, 682
 - assignments, 726-728
 - classes
 - hierarchies, 650
 - inheritance, 714
 - dictionaries, 671, 730-732
 - slots and, 826-828
 - generating, 582-583
 - global, 725
 - global statements, 401-402
 - links, 732-733
 - modules
 - dictionaries, 582-584
 - __init__.py files, 603
 - nesting, 586-587
 - partitioning, 560
 - namespace trees, 718
 - object namespaces, 725-726
 - OOP, 653
- naming variables, 277-280
- nested calls, 57
- nested modules, 596
- nested patterns, match statements, 298
- nested scopes, 406-407
 - arbitrary nesting, 409
 - name assignments, 406
 - name references, 406
- nested statement blocks, 248
 - special cases, 253-254
- nesting, 241, 1164
 - assigning nested sequences, 265
 - classes, scope, 728-730
 - combined clauses, try statement, 896
 - composites, 695
 - decorators, 1040-1041, 1085
 - descriptors, 1001
 - dictionaries, 70-71, 198
 - exception groups, 929
 - exception handlers, 927
 - control-flow, 928-930
 - syntactic nesting, 930-931
 - for loops, 322-323
 - functions, decorating methods, 1050
 - generator expressions, 498
 - imports, 1179
 - interactive loops, 259
 - iterators, 344
 - lambda expression, 475
 - loops, multiple, 931-932
 - map function, 498
 - match statements, 296
 - namespaces, in modules, 586-587

- package folders, 595
- while loops, 315
- nesting objects, 66-67
- .NET Framework for Windows, IronPython, 22
- newline character, 132, 216, 974
- __next__ method, 746
- nonclass instances, 822
 - inheritance, 823
 - metaclasses, 1101-1103
- None object, 78, 232
- nonempty data structures, 231
- nonlocal statement, 382, 409-412
 - state retention and, 413
- normalization, Unicode, 987-989
- Notepad, BOM (byte order marker), 984-985
- Nuitka, 24
- NULL character, 134
- Numba, 10, 23, 26
- number types, files, 224
- numbers, 81
 - comparisons, 229
 - complex, 56
 - decimals, 56, 168
 - floating-point, 56, 114
 - floating-point numbers, 168
 - fractions, 168
 - integers, 56, 168
 - mathematical operations, 56
 - rational, 56
 - sets, 56
 - true and false, 231
 - types, 81
- numeric programming, 10
- NumPy, 4, 7, 10

O

- obfuscation, list comprehensions and, 489
- object class
 - as superclass, 822
 - inheritance type class, 825
- object factories, 682, 795-796
- object indexes, loops, 312
- object memory, 71
- object model
 - classes, user-defined, 821-822
 - instances, metaclasses, 822
- object namespaces, 725-726
- object serialization, 220
- object storage, conversions, 218-220
- object types, 12, 55, 81
 - (see also types)
 - built-ins

- cyclic data structures, 237
- extending, 817-821
- immutable types, 237
- references, 235
- repetition, 236
- hierarchy, 233
- object-based code, 649
- object-oriented language, 5, 11
- object-oriented, definition, 80
- object-relational mappers, 9
- objects, 53, 117
 - aggregation, 697
 - attributes, 358
 - name strings, 358
 - type names, 359
 - Booleans, 78
 - built-in, 54-55
 - bytearray, 955, 970-972
 - bytes, 949, 954
 - calling, core object types, 55
 - categories, 224
 - class objects, 652, 661
 - default behavior, 662
 - classes, docstrings, 361
 - comparisons, 228-231
 - core object types, 229
 - copies, 226-228
 - copying, 122
 - databases, storage in, 704
 - pickle module, 704
 - shelf object update, 708
 - shelve database, 705-707
 - shelve module, 704, 705
 - dynamic typing, 117
 - embedding, 696
 - expressions, 53
 - file objects, 213
 - first-class object model, 465-466
 - flexibility, 225-226
 - garbage collection and, 118-119
 - generator objects, 496, 500, 752-756
 - implementation-related, 55
 - in-place changes, 121-123
 - instance objects, 652, 661, 662
 - attributes, 680
 - iterable (see iterable objects)
 - method objects, 791-794
 - bound methods, 791
 - plain functions, 791
 - mutable
 - changing in place, 224
 - functions, 550-552

- namespace objects, 682
- nesting, 66-67
- None, 78, 232
- operation sharing, 224
- program units, 55
- properties, class statement, 834
- Pydoc, 63
- reference counts, 71
- references, 226-228
 - assignments, 261
 - dictionaries, 185
- sets, 77-78, 1185
- shared objects, 120
- slice objects, 743
- state information, 467
- storage
 - CSV module, 222
 - JSON, 221-222
 - pickle module, 220-221
 - struct module, 222
- str, 954
- type objects, 78-79
- types, 118
- offsets, 331-332
- OO (object-oriented) code, 649
- OO (object-oriented), definition, 3
- OOP (object-oriented programming), 3, 649
 - arguments, functions, 650
 - attribute fetches, 651-653
 - class objects, 661
 - default behavior, 662
 - class statement, 80
 - class trees, 654-656, 1186
 - classes, 652, 679
 - constructors, 680-681
 - inheritance, 664-666
 - intercepting operators, 667-670
 - methods, 683-686
 - methods, augmenting, 689-691
 - subclasses, 665
 - subclassing, 688-690
 - superclasses, 652, 665
 - testing, 681-682
 - code reuse, 656-659, 870
 - composition, 650, 782-786
 - consistency, 870
 - delegation, 786-788
 - encapsulation, 779, 870
 - frameworks, 659
 - inheritance, 650, 779-782
 - attributes, 651-653
 - multiple inheritance, 654
 - search attribute, 650
 - instance objects, 662
 - instances, 650, 652, 661
 - maintenance, 870
 - method calls, 653-654
 - methods, 655, 663
 - namespaces, 653
 - operator overload, 650, 656
 - operator overloading, 686
 - print displays, 686-687
 - polymorphism, 657-658, 691-692, 779, 870
 - reloading, 661
 - search attribute, 650
 - structure, 870
- open function, 77, 400
 - customizing, 421
- operations
 - dictionaries, 185
 - mapping, 69-70
 - strings, 138-139
- operator overloading, 225, 668, 686, 739, 1182
 - attribute privacy, 761-762
 - attributes
 - assignment, 760-761
 - deleting, 760-761
 - references, 759
 - Boolean tests, 775
 - call expressions, 770-773
 - class type propagation, 767-769
 - comparison operators, 773
 - constructors, 740
 - __contains__ method, 756-758
 - descriptors, 1073-1075
 - destructors, 776-777
 - display formats, 762-765
 - expressions, 740
 - fallback options, 756-758
 - __getattr__ method, 1008
 - __getattribute__ method, 1008
 - __iadd__ method, 770
 - in-place addition, 770
 - index iteration, 745-746
 - instance indexing, 742-745
 - iterable objects, 746-747
 - multiple, 749-751
 - user-defined iterables, 747-749
 - metaclass methods, 1120
 - methods, 740-742
 - inline coding, 1071-1072
 - superclasses, 1072-1073
 - object comparisons, 773-774
 - OOP, 650, 656

- print displays, 686-687
- `__radd__` method, 766
- `__add__` reuse, 767
- results, 670
- right-side addition, 766-769
- strings, 139
- super function, 858-859
- operators
 - generic operations, 1163
 - lists, 175
- optimization tools, 943
- ordinals, 145
- ORMs (object-relational mappers), 9
- os module, 45
 - descriptor files, 224
- OSError exception class, 917
- output buffering, 214
- overloading operators, 225

P

- package-relative imports, 593
 - absolute, 604, 605
 - name clashes, 638
 - relative, 604-608, 609
- packages
 - benefits, 600-602
 - dotted-path syntax, 594
 - module import, 596
 - relative imports, 605
 - encapsulation, 779
 - folder bundles, 593
 - folders
 - names, 595
 - nesting, 595
 - imports, 593, 594, 609, 1180
 - folder hierarchies, 594
 - from statement, 596
 - modules, 596-597
 - `__init__.py` file, 597-598, 603
 - `__init__.py` files
 - initialization, 603
 - `__main__.py` file, 599-600
 - module search path and, 594
 - name clashes, 638
 - namespace packages, 593, 598, 609-613
 - module searches, 610-611
 - regular packages, 598
- pandas, 10
- parallel programming, async functions, 518
- parameters, functions, 383
- parentheses, 372
 - function calls, 373
 - lambda expression, 471
 - statements, 249
 - tuples, 209-210
- parsing
 - slicing, 140
 - text, string methods, 150
- pass statements, 313
- pathnames, 980-983
- patterns
 - factory functions, 407-409
 - re module, 977
 - sequence assignments, 265
 - structural pattern matching, 295
- per-call scopes, 395
- performance, 20
 - list comprehensions, 489
- permutation, sequences, 509-513
- persistence, 786
- pickle module, 238, 704, 978-979
 - object storage, 220-221
- pickle objects, 9, 786
- pipes, 224
- plain functions, 791, 838
- plain-function methods, 838-839
- pointers, 117
- polymorphism, 59, 79, 82, 386-387, 657-658, 691-692, 779, 1167
 - dynamic typing, 124
 - interfaces and, 780
 - intersect function, 389
 - strings and, 139
- popularity rise of Python, 1127
- portability, 4, 11
- Portable Operating System Interface (POSIX), 8
- positional arguments, 1076-1078
- positionals, argument matching, 428
- POSIX (Portable Operating System Interface), 8
- precision in large numbers, 56
- print emulator, keyword-only arguments, 449
- print function, 283-284
 - argument matching, 448-451
 - call format, 282-283
 - end-of-line character, 284
 - f-strings, 284
 - keyword arguments, 284
- print operations, 282
 - file object methods, 282
- print statement, 289
- print stream redirection
 - automatic, 286-288
 - hello world, 285
 - manual, 285-286

- standard output streams, 282
- stdout, 289
- print statements, 18
- print stream
 - redirecting
 - automatic, 286-288
 - hello world, 285
 - manual, 285-286
- print, viewing docstrings, 361
- printing
 - exceptions, built-in, 918-919
 - custom printing, 919-920
 - operator overloading and, 686-687
- procedures, 380
- productivity, developers, 3, 5
- profilers, 943
- program architecture
 - attributes, 561-563
 - files
 - text files, 561
 - top-level, 561
 - modules
 - files, 561
 - importing, 561-563
 - standard-library, 563
- program design (see design)
- program execution, 18
- program files
 - command line, 34-36
 - file icons, 37
 - modules, 34
 - naming convention, 18
 - scripts, 34
- program units, 55
- programming-in-the-large, 12
- programs, 1157
 - exercise, 48
 - module files, 34
 - modules, 53
- prompts, interactive coding, 30, 32-34
- properties, 834, 993, 1015
 - attribute fetches, 993
 - attributes, 835
 - computing value, 995
 - creating, 993
 - decorators, 996
 - deleters, 996
 - setters, 996
 - descriptors and, 1006-1007
 - inheritance, 994
 - objects, class statement, 834
- prototyping, 9
- pseudoprivate attributes, 788-791
- pseudoprivate naming, 804
- PVM (Python Virtual Machine), 20, 26
 - .py suffix, 575
- py2wasm, 9
- .pyc files, 19
- PyCharm, 39
- PyDev, 39
- Pydoc, 63, 357
 - browser mode, 366-368
 - built-in tools, 363-365
 - imported modules, 369
 - module-index page, 367
 - modules, 365-366
- Pygments, 370
- Pyjamas, 9
- pyjnius, 9
- Pyodide, 9
- PyPy, 7, 10, 22, 23, 26
- PyQt, 8
- PyScript, 9
- PyScripter, 39
- Pyston, 24
- Python
 - on Android, 1148-1151
 - applications, 8-10
 - bytecode and, 6
 - code size, 3
 - current users, 7
 - developer productivity, 3, 5
 - embedding, 45
 - execution speed, 6
 - on iOS, 1151-1152
 - on Linux, 1146-1148
 - on macOS, 1141-1146
 - popularity rise, 1127
 - portability, 4
 - as scripting language, 5-6
 - software quality, 3, 4
 - on Windows, 1135
 - Command Prompt, 1137-1138
 - IDLE, 1139-1141
 - WSL, 1136
 - Python Virtual Machine, 20, 26
 - Python/Java bridges, 9
 - PYTHONPATH directory, 568
 - configuration search path, 570
 - PyThran, 10, 23
 - pywin32, 9
 - PyYAML, 9

Q

- qualification, attribute names, 584-585
- quality assurance applications, 10
- queues
 - FIFO (first-in-first-out), 461
 - recursion, 461-463
- quote characters in strings, 63

R

- `__radd__` method, 766
- `__add__` reuse, 767
- raise statement, 898, 932
 - except block, scopes, 899-900
 - except clause, 898-899
 - exception propagation, 900
 - from clause, 900-902
- random module, 57
- range object
 - counter loops, 324-325
 - iteration protocol, 342
 - lists, 327-328
 - sequence reordering, 326
 - sequence scans, 325-326
 - skipping items, 327
- range-testing decorator, 1076-1087
- rational numbers, 56
- raw bytes, 961
- raw strings, 448
 - docstrings, 360
 - escape sequences and, 136-137
- re module, pattern-matching, 977
- read-only descriptors, 999-1000
- readability, 3
- record factories, 683
- recursion, 228, 1181
 - arbitrary structures, 460
 - testing, 460-461
 - cycles, 463
 - direct, 459
 - from imports, 642-643
 - functions, 1175
 - if else expression, 458-459
 - indirect, 459
 - local variables, 395
 - versus loop statements, 459-460
 - paths, 463
 - queues, 461-463
 - stack limits, 463
 - stacks, 461-463
 - sum function, 457-458
- reduce function, 478-479

- redundancy
 - classes, 649
 - increase in, 1125
 - modules, 560
 - removing, 684
- reference counts, 71
- references, 226-228
 - built-in types, 235
 - lists, 174
 - names, 396
 - nested scopes, 406
 - shared, 120
 - argument passing, 424-426
 - equality, 123-124
 - in-place changes, 121-123
 - multiple-target assignments, 271-272
 - variables, 117
- relative imports, 604, 609
- relative magnitude comparisons, 229
- reload function, 587-590
 - from imports, 640
 - interactive testing and, 641
- reloading modules, 630
 - OOP, 661
 - recursive coding, 635-636
 - recursive reloaders, 631-632
 - testing, 632-635
 - reload variants, 636-638
- repetition, 54
- replace method, 148, 149
- REPLs (real-eval-print loops)
 - code folders, 29-30
 - f-string literal, 163
 - IPython, 30
 - preloading tools, 629
 - starting, 28-29
- `__repr__` method, 763
- reserved words in variable naming, 277
- REST, 9
- reStructuredText markup language, 370
- return statement, 381, 383
- returns, none, 552
- runtime declarations, 1034
 - function decorators, 1034
- runtime operations
 - imports, 564
 - module imports, 579

S

- sandbox, 1127
- scientific programming, 10
- SciPy, 10

- scopes, 393, 863-864
 - argument defaults and, 416-419
 - built-ins, 399
 - LEGB rule, 400-401
 - comprehension variables, 504-505
 - comprehensions, loop variables, 397
 - cross-file changes, 404-405
 - enclosing, 552, 1036
 - functions, 552
 - state retention and, 1067
 - examples, 398-399
 - exception variables, 397
 - exceptions, 899-900
 - global scopes, 394
 - global statement emulation, 405-406
 - global variables, program design, 402-403
 - global, state retention and, 413
 - lexical scoping, 394, 586
 - local scopes, 394
 - modules
 - imports and, 583
 - variables, 585-586
 - named assignments, 398
 - nested class, 728-730
 - nested scopes, 406-407
 - arbitrary nesting, 409
 - name assignments, 406
 - name references, 406
 - per-call scopes, 395
- script1.py, 34-35
- scripting
 - server-side scripting, 9
 - web scripting, 8
- scripting language, 5
 - control language and, 6
 - ease of use and, 6
 - shell tools and, 5
- scripts, 34, 1158
 - exercise, 48
 - web servers, 45
- search tables, 54
- searches
 - __init__.py files, 603
 - lists, 170
 - packages, 594
- selection, 54
- __self__ attribute, 789
- semicolons in statements, 239, 249
- send method, 494
- sequence, 54
- sequence assignments, 262, 264-266
- sequence operations, 145, 146, 170
 - arbitrary expressions, 58
 - byte strings, 967-968
 - indexing expressions, 58
 - list comprehensions, 210
 - lists, 65
- sequence patterns, 297
- sequence scans
 - for loop, 325-326
 - range object, 325-326
 - while loop, 325-326
- sequences, 82
 - dictionaries, 196
 - for loops, 319-321
 - lists, 168
 - permutation, 509-513
 - reordering, 326
 - slicing, 506
 - functions, 507
 - generator expressions, 508
 - generator functions, 507-508
 - tester, 509
- strings, 130, 168
- tuples, 168, 209
- virtual sequences, 336
- zip object, 328-330
- serialization
 - json module, 978-979
 - objects, 220
 - pickle module, 978-979
- server-side scripting, 9
- set comprehensions, 504
 - syntax, 485
- set function, 77
- __set__ method, 992
- __setattr__ method, 760-761, 992
- __setitem__ method, 742, 744
- sets, 56, 77-78, 81, 224
 - comparisons, 229
 - dictionaries, 200-201
- shared objects, 120
- shared references, 120
 - argument passing, 424-426
 - equality, 123-124
 - in-place changes, 121-123
 - multiple-target assignments, 271-272
- Shed Skin, 23, 26
- shell programs
 - command line, 36
 - stream redirection, 36
- shell tools, 5, 8
- shell-command streams, 224
- shelve module, 704, 705

- object storage, 705-707
 - updates, 708
- shelves, 224, 786
- shipping, 943
- single-scan iteration, 748
- site-packages directory, 569
- slice assignments, 170, 177-178, 184
- slice expressions, 743-744
- slice objects, 743
- slices, 751
 - cleanup and, 143
 - skipping items, 327
- slicing, 58, 130, 240, 1161
 - lists, 176-177
 - out of bounds, 240
 - parsing, 140
 - sequences, 506
 - functions, 507
 - generator expressions, 508
 - generator functions, 507-508
 - tester, 509
 - strings, 140-143
 - extended slicing, 142
- slots, 672, 732, 826
 - class-level defaults, 831
 - declaring, 826
 - descriptors, 1007
 - generic programs, 830
 - ListTree class, 832-833
 - multiple inheritance, 831
 - multiple inheritance classes, 833
 - namespaces dictionaries and, 826-828
 - single-inheritance trees, 832
 - speed, 833
 - subclasses, 831
 - superclasses, 831
 - usage rules, 827
- __slots__ attribute, 829
- smartphone apps, 10, 39
- SOAP, 9
- sockets, 224
- software quality of Python, 3, 4
- software reuse, 3
- spaces, indentation and, 252
- special characters, escape sequences, 132-135
- special methods, 837
 - (see also static methods)
- special-case syntax, 298
- Sphinx, 370
- split method, 150
- Spyder, 39
- SQLAlchemy, 9
- SQLite, 9
- SQLObject, 9
- Stackless, 23
- standalone executables, 24, 45, 1152-1155
- standard library, 4, 11
 - directories, module search path, 569
 - methods, 147
 - modules, 563
- standard manuals, 370-371
- standard output streams, print operations, 282
- standard streams, 223
- startup-speed optimization, bytecode and, 19
- state
 - descriptors, 1003-1006
 - event handlers, 772
 - exception classes, 911
 - exceptions, built-in, 918-919
 - custom, 920-922
 - instances, 1003
 - retaining, 1045-1048
 - classes, 416
 - function attributes, 414-415
 - global scope, 413
 - nonlocal statement, 413
 - scope mutables, 415
 - suspension, 490-491
- state information, function objects, 467
- statements, 245-247
 - as expression-based equivalents, 473-475
 - assignment statements, 247
 - assignments, 262
 - async, 382
 - await, 382
 - class, 80, 649, 713
 - attributes, 714-716
 - syntax, 713-714
 - compound statements, 72, 248, 291
 - concept hierarchy, 245
 - def, 381, 383, 384, 388
 - executable
 - from, 579
 - import, 579
 - expression statements, 280-281
 - expressions in, 53
 - for, 139
 - from, 577
 - potential problems, 580-582
 - from *, 577-578
 - function-related, 379
 - global, 382
 - namespaces, 401-402
 - if statements, 248, 291

- multiple-choice selections, 293-295
- if/then/else, 306-307
- import, 576-577
- indentation, 250-252
- interactive loops, 254-259
- match statements, 72, 295
 - attribute patterns, 298
 - if statements, 296
 - instance patterns, 298
 - literal patterns, 297
 - mapping patterns, 297
 - nested patterns, 298
 - nesting, 296
 - parentheses, 298
 - sequence patterns, 297
- module imports, 582
- nested statements
 - blocks, 248
 - special cases, 253-254
- nonlocal, 382, 409-412
- order, 638-639
- parentheses, 249
- raise, 932
- return, 381, 383
- semicolon, 239
- semicolons, 249
- special cases rules, 252-253
- syntax, 246, 299
 - blocks, 300
 - delimiters, 302
 - indentation, 300-302
 - special syntax, 303-304
- truth values, 304-306
- try, 257-258
- yield, 382, 491
- static methods, 717, 837, 838, 840
 - alternatives, 839
 - instance counting, 841-842
- staticmethod function, 840
- statistically nested scopes, 406
 - (see also nested scopes)
- stdout, 289
- STEM (science, technology, engineering, and math), 4
- storage
 - class, 864
 - objects
 - CSV module, 222
 - JSON, 221-222
 - pickle module, 220-221
 - struct module, 222
- per-instance, 864
- str function, 57
- __str__ method, 763-765
- str object, 954
- stream processors, composition, 784
- streams
 - redirection, shells, 36
 - shell command, 224
- string formatting, 151, 968
 - expressions, 152
 - advanced, 155
 - custom formats, 154-155
 - dictionary-based, 156-157
 - format method, 152, 157
 - advanced, 160-162
 - attributes, 158
 - binary formats, 161
 - custom formats, 159-160
 - f-string literal, 162-167
 - floating-point numbers, 161
 - formatspec component, 159-160
 - hex formats, 161
 - keys, 158
 - octal formats, 161
 - offsets, 158
 - substitution targets, 159
 - literals, 152
 - manual, 152
 - __repr__ method, 762-765
 - __str__ method, 762-765
- string indexing, 241, 1163
- string interpolation, 163
- string literals, 956-957
 - backslash escape sequences and, 63
 - docstrings, 360
 - format, 131
 - operations, 130
 - quotes, 131-132
 - UTF-8 encoding, 964
- string methods, 146, 148
 - find, 149
 - join, 149
 - method calls, syntax, 146
 - parsing text, 150
 - replace, 148, 149
 - split, 150
 - substring test, 151
 - whitespace, 151
- string objects, 898
- string representation methods, 762-765
- strings, 81, 129, 168, 215
 - as immutable sequences, 130
 - backslash characters, 133-134

- backslash escape, 63
- block strings, 137-138
- character arrays and, 129
- code points, 133
- combining, 130
- comparisons, 145, 229
- concatenation, 130, 132, 138
- conversion, 143
 - character-code, 144
 - floating-point numbers, 144
 - strings to numbers, 143
- delimiter, 149
- docstrings, 138
- empty, 216
- escape characters, 132-135
- expressions, 130
- fetching by offset, 130
- help resources, 62-63
- immutability, 59-60
- indexing, 130, 140-143
- iteration, 139
- loops and, 139
- methods, 60-62, 130
- modules, 130
- multiline, 137-138
- newline characters, 132
- NULL character, 134
- operations, 138-139
- operator overload and, 139
- polymorphism and, 139
- quote characters, 63
- raw, 136-137
- section extraction, 130
- sequence operations, 145-146
 - arbitrary expressions, 58
 - indexing expressions, 58
- sequences, 130
- slicing, 130, 140-143
 - extended slicing, 142
- split method, 219
- strip method, 219
- substrings, testing for, 151
- text strings, 64
- triple-quotes, 137-138
- Unicode, 64, 129
- struct module, 238
 - binary data, 978
 - object storage, 222
- structural pattern matching, 295
- __sub__ method, operator overloading, 740
- subclasses, 665
 - metaclasses, 1094
 - slots, 831
- subclassing, 688, 1183
 - built-in types extension, 819-821
 - extending code, 692
 - hierarchies, 693
 - inheritance, 692, 823
 - instances, namespace dictionaries, 731
 - methods, 689-690
 - polymorphism, 691-692
- subprime code, 317
- subroutines, 380
- substrings, testing for, 151
- sum function, recursion and, 457-458
- super built-in function, 691
- super function, 851, 1116
 - argument lists, 857-858
 - attribute-fetch algorithm, 853
 - call-chain anchors, 855-857
 - cooperator method dispatch, 855-857
 - deployment, 854
 - MRO algorithm, 852-853
 - noncalls, 858-859
 - operator overloading, 858-859
- superclasses, 652, 665
 - abstract, 722-724, 784
 - library tools, 723
 - constructors, 694, 864
 - exception superclass, 913
 - exceptions, 912
 - interfacing with, 721-722
 - metaclasses comparison, 1110-1111
 - method calls, 719
 - operator overloading method coding, 1072-1073
 - slots, 831
 - __slots__ attribute, 829
- support, 4
- SWIG, 9
- syntactic nesting, exception handlers, 930-931
- syntax
 - argument matching, 429-430
 - assignments, 270
 - class statement, 713-714
 - combined clauses, try statement, 895
 - compound statements, 248
 - comprehension syntax, 483
 - decorators, 1033
 - dictionary comprehensions, 485
 - dotted-path, 594, 596
 - relative imports, 605
 - function decorator, 836
 - function decorators, 1034
 - instance generation, 821

- lambda expression, 471
- list comprehensions, formal, 485
- method calls, 146
- set comprehensions, 485
- special-case, 298
- statements, 246, 299
 - block rule special cases, 253-254
 - blocks, 300-302
 - delimiters, 302
 - indentation, 249-252, 300-302
 - nested statement blocks, 248
 - parentheses, 249
 - semicolons, 249
 - special cases rules, 252-253
 - special syntax, 303-304
- tuples, 209-210
- variable names, 277
- sys.exc_info, 935-937
- sys.platform, 35
- systems programming, 8

T

- tabs, indentation and, 252
- templates, 9
 - replacements, 148
- termination actions
 - try/finally and, 893
 - with statement, 904
- termination handlers, 908-909
- testdriver function, 935
- tester, slicing sequences, 509
- testing, 10
 - class creation, 681-682
 - comparisons, 228-231
 - equality, 229
 - in-process tests, exceptions, 935
 - interactive tests, 241
 - user input, error handling, 256-257
- testing tools, 942
- text editors, 45
 - BOM (byte order marker), 984-985
 - indentation, 252
- text files, 217-218, 955-956
 - binary mode, 973
 - bytes object, 974
 - conversions, 218-220
 - iterators, 215
 - lines comparison, 908
 - pathname, 973
 - text mode, 973
 - newline character, 974
 - Unicode text files, 975-977
- text output file, 75
- text strings
 - literals, 956-957
 - non-ASCII characters, 64
 - type conversions, 957-959
 - Unicode code-points, 64
 - Unicode text, 959-963
- text-mode files
 - BOM headers, 972
 - newline character, 974
- textual data, 954
- third-party domain, 4
- third-party utilities, 12
- timeit module
 - API-calls mode, 540
 - benchmarking automation, 544
 - benchmark module, 544-545
 - benchmark script, 545-546
 - dictionaries, 547-548
 - individual Pythons, 546-547
 - multiple Pythons, 547
 - sets, 547-548
 - callable objects, 540
 - command-line mode, 541-542
 - multiline statements, 542-543
 - setup code, 543
 - sort speed, 543-544
 - statement strings, 540
- timer decorator, 1057
- timer utility functions, 531
- timestamp-based bytecode files, 565
- timing tools, 1173
- Toga, 8, 1154
- tools
 - built-in, 12
 - increase in, 1125
- toolsets
 - built-in tools, 941
 - development tools, 941
 - debuggers, 943
 - documentation tools, 942
 - error-checking tools, 942
 - IDEs, 942
 - installation management, 944
 - optimization, 943
 - profilers, 943
 - shipping, 943
 - testing tools, 942
 - extensions, 941
- traceback object, 937
- triple-quoted strings, 137-138
- truth values, 231-233, 304-306

- try statement
 - built-in exceptions, 891
 - combined clauses, 894
 - nesting, 896
 - syntax, 895
 - debugging, 934-935
 - default exception behavior, 890
 - else clause, 886-892
 - error handling, 257-258
 - except clause, 886-892
 - except* clause, 923-925
 - finally block, 894-895
 - finally clause, 892-894
 - forms, 886
 - handlers, 885
 - nesting, 927-928
 - tuples, 887
- try/finally, 894-895
 - nesting, 896
 - syntax, 895-896
 - termination actions and, 893
- tuples, 74-75, 81, 168, 183, 197, 207-208, 1162
 - assignments, 262
 - commas, 209-210
 - commas in expressions, 239
 - compared to lists, 211
 - conversions, 210-211
 - converting from dictionaries, 212
 - for loops, 319-321
 - generating, 238
 - immutability, 210-211
 - methods, 210-211
 - named, 212-213
 - parentheses, 209-210
 - sequence assignment, 209
 - sorting, 210
 - syntax, 209-210
 - try statement, 887
- TurboGears, 9
- type calls, comprehensions, 504
- type conversions, text strings, 957-959
- type function, 78-79
- type hinting, 79, 125-126
- type objects, 78-79, 233
 - built-ins, 235
- type-specific operations, lists, 65-66
- types, 12
 - built-ins, extending, 818
 - classes and, 821
 - immutable, 169, 241
 - mappings, 168
 - metaclass/class dichotomy, 824-825
 - metaclasses
 - class statements, 1095-1097
 - instances, 1093-1094
 - subclasses, 1094-1095
 - mutable, 169
 - numbers, 168
 - object types, 118
 - hierarchy, 233
 - sequences, 168
- types library module, 233
- TypeScript, 79
- typing, dynamic, 12

U

- UIs (user interfaces), 8
- Unicode, 949
 - character encodings, 951-954
 - character representations, 950-951
 - code points, 133, 951
 - value, 960
 - code-points, 64
 - default encoding, 982
 - escapes, 960
 - files, 76-77
 - normalization, 987-989
 - raw bytes, 961
 - strings, 64, 129
 - code points, 64
 - strings, coding, 959-963
 - text files, 975-977
 - UTF-16, 133
- union operation, dictionaries, 193, 194
- unit testing, __name__ attribute, 623-625
- Unix, Python installation, 27
- unpacking assignments, 263, 266-268
 - boundary cases, 268-269
 - for loops, 270
- update method, 71
- usage mode flags, 623
- user inputs
 - error handling, 256-257
 - math, 255-256
- user interfaces (UIs), 8
- user-defined classes, 821-822
- user-defined docstrings, 360-362
- user-defined exceptions, nonerror conditions, 933
- user-defined function decorators, 847-848
- user-defined iterables, 747-756
- UTF-16 encoding, 133, 953, 960
- UTF-32 encoding, 953, 960
- UTF-8 encoding, 952, 982
 - BOM (byte order marker), 985-987

- string literals, 964
- utilities
 - library utilities, 12
 - third-party, 12

V

- values method, 73
- variables, 32
 - __all__, 619-620
 - comprehensions, 397
 - scopes and, 504-505
 - creating, 116
 - from * statement, 640
 - global
 - coupling, 455
 - program design, 402-403
 - state, decorators and, 1046
 - imports, 585-586
 - local, 389-390
 - static locals, 467
 - loops, 552
 - names, 576
 - names, case, 277
 - naming rules, 277-280
 - references, 117
 - scope, 394
 - scopes, 585-586
 - types, 116
 - use, 116
- view objects, 199-200
- virtual machines (VMs) (see VMs (virtual machines))
- virtual sequences, 336
- visibility, class attributes, 715
- VMs (virtual machines), 19
 - PVM (Python Virtual Machine), 20
- VSCode, 39

W

- Wasm (WebAssembly), 40
- web scripting, 8
- web server scripts, 45
- WebAssembly, 8, 9
- while loops, 311-312
 - break statement, 315-316
 - named assignment, 315
 - break statements, 313
 - continue statement, 314-315
 - nested code, 315
 - do until, 312
 - else, 311

- else clause, 316-317
- interactive loops, 254
- pass statement, 313-314
 - ellipsis, 314
- sequence scans, 325-326
- whitespace, 151, 250
- wide-character strings, 951
- Windows
 - directory paths, 217
 - interactive coding and, 28
 - Python installation, 27, 1136
 - Python on, 1135
 - Command Prompt, 1137-1138
 - IDLE, 1139-1141
 - WSL, 1136
- Wing, 39
- with statement, 904-905
 - context managers, multiple, 907-908
 - context-management protocol, 905-907
 - termination actions, 904
 - termination handlers, 908
- working directory, commands, 36
- wrappers, 787, 938
 - call proxies, 1032
 - class decorators, 1032
 - layers, 1041
 - stacking, 1040
- wrapping code, 865
- WSL (Windows Subsystem for Linux), 1136
- wxPython, 8

X

- _X prefix, 619-620
- __X pseudoprivate name mangling, 788, 1070
- XML-RPC, 9

Y

- yield function, generator objects, 752-756
- yield statement, 382, 491
 - generator functions, 494-495

Z

- zip function
 - emulating, 513-517
 - iterables, 515
- zip object, 328-330
 - iteration protocol, 343
- ZODB, 9
- Zope, 9

About the Author

Mark Lutz is the author of Python's classic and foundational texts, a former trainer with two decades of experience teaching Python to newcomers, and one of the people responsible for the prominence that Python enjoys today.

Mark wrote the three O'Reilly books *Learning Python*, *Programming Python*, and *Python Pocket Reference*, all currently in fourth, fifth, or sixth editions. He has been using and promoting Python since 1992, started writing Python books in 1995, and began teaching Python classes in 1997.

All told, Mark has taught thousands of learners live and in person, and his published works span 12k pages among 15 books that cover Pythons 1.X through 3.X. He also has BS and MS degrees in computer science, work experience in compilers and other domains, and a current interest in Python-coded apps that run on both PCs and phones.

For more author background, see Mark's books-and-software website, learning-python.com.

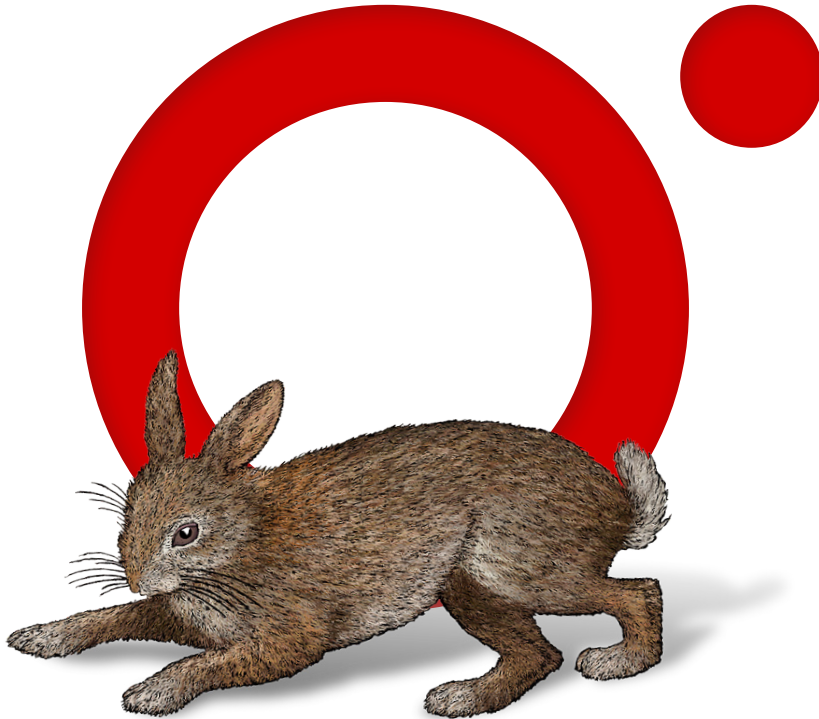
Colophon

The animal on the cover of *Learning Python*, sixth edition, is a wood rat (*Neotoma muridae*). The wood rat lives in a wide range of conditions (mostly rocky, scrub, and desert areas) over much of North and Central America, generally at some distance from humans. Wood rats are good climbers, nesting in trees or bushes up to six meters off the ground; some species burrow underground or in rock crevices or inhabit other species' abandoned holes.

These grayish-beige, medium-size rodents are the original pack rats: they carry anything and everything into their homes, whether or not it's needed, and are especially attracted to shiny objects such as tin cans, glass, and silverware.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a 19th-century engraving from *Cuvier's Animals*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

**Learn from experts.
Become one yourself.**

60,000+ titles | Live events with experts | Role-based courses
Interactive learning | Certification preparation

Try the O'Reilly learning platform free for 10 days.

