

The background of the cover features a complex 3D abstract design. It includes several metallic, reflective, and flowing shapes that resemble stylized waves or intertwined ribbons. These shapes are rendered in shades of gray and white, with highlights and shadows that give them a three-dimensional appearance. Overlaid on these shapes are several thin, intersecting lines in red, blue, and purple, creating a geometric pattern that adds to the overall complexity of the design.

Kenneth A. Lambert

Fundamentals of Python

First Programs

Fundamentals of Python:

First Programs

Kenneth A. Lambert
Martin Osborne, Contributing Author



COURSE TECHNOLOGY
CENGAGE Learning™

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

This is an electronic version of the print textbook. Due to electronic rights restrictions, some third party content may be suppressed. Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. The publisher reserves the right to remove content from this title at any time if subsequent rights restrictions require it. For valuable information on pricing, previous editions, changes to current editions, and alternate formats, please visit www.cengage.com/highered to search by ISBN#, author, title, or keyword for materials in your areas of interest.

Fundamentals of Python: First Programs
Kenneth A. Lambert

Executive Editor: Marie Lee

Acquisitions Editor: Brandi Shailer

Senior Product Manager: Alyssa Pratt

Development Editor: Ann Shaffer

Associate Product Manager: Stephanie
LorenzAssociate Marketing Manager: Shanna
Shelton

Content Project Manager: Jennifer Feltri

Art Director: Faith Brosnan

Image credit: © istockphoto/Pei Ling Hoo

Cover Designer: Wing-ip Ngan,
Ink design, Inc.

Compositor: GEX Publishing Services

© 2012 Course Technology, Cengage Learning

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at
Cengage Learning Customer & Sales Support, 1-800-354-9706

For permission to use material from this text or product, submit all
requests online at **www.cengage.com/permissions**
Further permissions questions can be emailed to
permissionrequest@cengage.com

Library of Congress Control Number: 2011920241

ISBN-13: 978-1-111-82270-5

ISBN-10: 1-111-82270-0

Course Technology20 Channel Center
Boston, Massachusetts 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at:
international.cengage.com/region

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **course.cengage.com**.

Purchase any of our products at your local college store or at our preferred online store **www.cengagebrain.com**.

Some of the product names and company names used in this book have been used for identification purposes only and may be trademarks or registered trademarks of their respective manufacturers and sellers.

Any fictional data related to persons or companies or URLs used throughout this book is intended for instructional purposes only. At the time this book was printed, any such data was fictional and not belonging to any real persons or companies.

Course Technology, a part of Cengage Learning, reserves the right to revise this publication and make changes from time to time in its content without notice.

The programs in this book are for instructional purposes only. They have been tested with care, but are not guaranteed for any particular intent beyond educational purposes. The author and the publisher do not offer any warranties or representations, nor do they accept any liabilities with respect to the programs.

Printed in the United States of America

1 2 3 4 5 6 7 15 14 13 12 11

Table of Contents

[CHAPTER]	1	INTRODUCTION	1
	1.1	Two Fundamental Ideas of Computer Science: Algorithms and Information Processing	2
		1.1.1 Algorithms	2
		1.1.2 Information Processing	4
	1.1	Exercises	5
	1.2	The Structure of a Modern Computer System	6
		1.2.1 Computer Hardware	6
		1.2.2 Computer Software	8
	1.2	Exercises	10
	1.3	A Not-So-Brief History of Computing Systems	10
		1.3.1 Before Electronic Digital Computers	11
		1.3.2 The First Electronic Digital Computers (1940–1950)	15
		1.3.3 The First Programming Languages (1950–1965)	16
		1.3.4 Integrated Circuits, Interaction, and Timesharing (1965–1975)	18
		1.3.5 Personal Computing and Networks (1975–1990)	19
		1.3.6 Consultation, Communication, and Ubiquitous Computing (1990–Present)	21
	1.4	Getting Started with Python Programming	23
		1.4.1 Running Code in the Interactive Shell	23
		1.4.2 Input, Processing, and Output	25
		1.4.3 Editing, Saving, and Running a Script	28
		1.4.4 Behind the Scenes: How Python Works	29
	1.4	Exercises	30
	1.5	Detecting and Correcting Syntax Errors	31
	1.5	Exercises	32
		Suggestions for Further Reading	32
		Summary	33
		Review Questions	35
		Projects	37
[CHAPTER]	2	SOFTWARE DEVELOPMENT, DATA TYPES, AND EXPRESSIONS	39
	2.1	The Software Development Process	40
	2.1	Exercises	43
	2.2	Case Study: Income Tax Calculator	43
		2.2.1 Request	43
		2.2.2 Analysis	44
		2.2.3 Design	44
		2.2.4 Implementation (Coding)	45
		2.2.5 Testing	46

2.3	Strings, Assignment, and Comments.....	47
2.3.1	Data Types.....	47
2.3.2	String Literals.....	48
2.3.3	Escape Sequences.....	50
2.3.4	String Concatenation.....	50
2.3.5	Variables and the Assignment Statement.....	51
2.3.6	Program Comments and Docstrings.....	52
2.3	Exercises.....	53
2.4	Numeric Data Types and Character Sets.....	54
2.4.1	Integers.....	54
2.4.2	Floating-Point Numbers.....	55
2.4.3	Character Sets.....	55
2.4	Exercises.....	57
2.5	Expressions.....	58
2.5.1	Arithmetic Expressions.....	58
2.5.2	Mixed-Mode Arithmetic and Type Conversions.....	60
2.5	Exercises.....	63
2.6	Using Functions and Modules.....	63
2.6.1	Calling Functions: Arguments and Return Values.....	64
2.6.2	The math Module.....	65
2.6.3	The Main Module.....	66
2.6.4	Program Format and Structure.....	67
2.6.5	Running a Script from a Terminal Command Prompt.....	68
2.6	Exercises.....	70
	Summary.....	70
	Review Questions.....	72
	Projects.....	73

[CHAPTER] **3** CONTROL STATEMENTS **75**

3.1	Definite Iteration: The for Loop.....	76
3.1.1	Executing a Statement a Given Number of Times.....	76
3.1.2	Count-Controlled Loops.....	77
3.1.3	Augmented Assignment.....	79
3.1.4	Loop Errors: Off-by-One Error.....	80
3.1.5	Traversing the Contents of a Data Sequence.....	80
3.1.6	Specifying the Steps in the Range.....	81
3.1.7	Loops That Count Down.....	82
3.1	Exercises.....	83
3.2	Formatting Text for Output.....	83
3.2	Exercises.....	86
3.3	Case Study: An Investment Report.....	87
3.3.1	Request.....	87
3.3.2	Analysis.....	87
3.3.3	Design.....	88
3.3.4	Implementation (Coding).....	88
3.3.5	Testing.....	90
3.4	Selection: if and if-else Statements.....	91
3.4.1	The Boolean Type, Comparisons, and Boolean Expressions.....	91
3.4.2	if-else Statements.....	92

	3.4.3	One-Way Selection Statements	94
	3.4.4	Multi-way <code>if</code> Statements	95
	3.4.5	Logical Operators and Compound Boolean Expressions	97
	3.4.6	Short-Circuit Evaluation	99
	3.4.7	Testing Selection Statements	100
3.4		Exercises	101
3.5		Conditional Iteration: The <code>while</code> Loop	102
	3.5.1	The Structure and Behavior of a <code>while</code> Loop	102
	3.5.2	Count Control with a <code>while</code> Loop	104
	3.5.3	The <code>while True</code> Loop and the <code>break</code> Statement	105
	3.5.4	Random Numbers	107
	3.5.5	Loop Logic, Errors, and Testing	109
3.5		Exercises	109
3.6		Case Study: Approximating Square Roots	110
	3.6.1	Request	110
	3.6.2	Analysis	110
	3.6.3	Design	110
	3.6.4	Implementation (Coding)	112
	3.6.5	Testing	113
		Summary	113
		Review Questions	116
		Projects	118

[CHAPTER] 4

STRINGS AND TEXT FILES

121

4.1		Accessing Characters and Substrings in Strings	122
	4.1.1	The Structure of Strings	122
	4.1.2	The Subscript Operator	123
	4.1.3	Slicing for Substrings	124
	4.1.4	Testing for a Substring with the <code>in</code> Operator	125
4.1		Exercises	126
4.2		Data Encryption	126
4.2		Exercises	129
4.3		Strings and Number Systems	129
	4.3.1	The Positional System for Representing Numbers	130
	4.3.2	Converting Binary to Decimal	131
	4.3.3	Converting Decimal to Binary	132
	4.3.4	Conversion Shortcuts	133
	4.3.5	Octal and Hexadecimal Numbers	134
4.3		Exercises	136
4.4		String Methods	136
4.4		Exercises	140
4.5		Text Files	141
	4.5.1	Text Files and Their Format	141
	4.5.2	Writing Text to a File	142
	4.5.3	Writing Numbers to a File	142
	4.5.4	Reading Text from a File	143
	4.5.5	Reading Numbers from a File	145
	4.5.6	Accessing and Manipulating Files and Directories on Disk	146

4.5	Exercises.....	148
4.6	Case Study: Text Analysis.....	148
4.6.1	Request.....	149
4.6.2	Analysis.....	149
4.6.3	Design.....	150
4.6.4	Implementation (Coding).....	151
4.6.5	Testing.....	152
	Summary.....	153
	Review Questions.....	154
	Projects.....	156

[CHAPTER] 5 LISTS AND DICTIONARIES 159

5.1	Lists.....	160
5.1.1	List Literals and Basic Operators.....	160
5.1.2	Replacing an Element in a List.....	163
5.1.3	List Methods for Inserting and Removing Elements.....	165
5.1.4	Searching a List.....	167
5.1.5	Sorting a List.....	168
5.1.6	Mutator Methods and the Value <code>None</code>	168
5.1.7	Aliasing and Side Effects.....	169
5.1.8	Equality: Object Identity and Structural Equivalence.....	171
5.1.9	Example: Using a List to Find the Median of a Set of Numbers.....	172
5.1.10	Tuples.....	173
5.1	Exercises.....	174
5.2	Defining Simple Functions.....	175
5.2.1	The Syntax of Simple Function Definitions.....	175
5.2.2	Parameters and Arguments.....	176
5.2.3	The <code>return</code> Statement.....	177
5.2.4	Boolean Functions.....	177
5.2.5	Defining a <code>main</code> Function.....	178
5.2	Exercises.....	179
5.3	Case Study: Generating Sentences.....	179
5.3.1	Request.....	179
5.3.2	Analysis.....	179
5.3.3	Design.....	180
5.3.4	Implementation (Coding).....	182
5.3.5	Testing.....	183
5.4	Dictionaries.....	183
5.4.1	Dictionary Literals.....	183
5.4.2	Adding Keys and Replacing Values.....	184
5.4.3	Accessing Values.....	185
5.4.4	Removing Keys.....	186
5.4.5	Traversing a Dictionary.....	186
5.4.6	Example: The Hexadecimal System Revisited.....	188
5.4.7	Example: Finding the Mode of a List of Values.....	189
5.4	Exercises.....	190

5.5	Case Study: Nondirective Psychotherapy	191
5.5.1	Request	191
5.5.2	Analysis	191
5.5.3	Design	192
5.5.4	Implementation (Coding)	193
5.5.5	Testing	195
	Summary	195
	Review Questions	196
	Projects	198
[CHAPTER] 6	DESIGN WITH FUNCTIONS	201
6.1	Functions as Abstraction Mechanisms	202
6.1.1	Functions Eliminate Redundancy	202
6.1.2	Functions Hide Complexity	203
6.1.3	Functions Support General Methods with Systematic Variations	204
6.1.4	Functions Support the Division of Labor	205
6.1	Exercises	205
6.2	Problem Solving with Top-Down Design	206
6.2.1	The Design of the Text-Analysis Program	206
6.2.2	The Design of the Sentence-Generator Program	207
6.2.3	The Design of the Doctor Program	209
6.2	Exercises	210
6.3	Design with Recursive Functions	211
6.3.1	Defining a Recursive Function	211
6.3.2	Tracing a Recursive Function	213
6.3.3	Using Recursive Definitions to Construct Recursive Functions	214
6.3.4	Recursion in Sentence Structure	214
6.3.5	Infinite Recursion	215
6.3.6	The Costs and Benefits of Recursion	216
6.3	Exercises	218
6.4	Case Study: Gathering Information from a File System	219
6.4.1	Request	219
6.4.2	Analysis	220
6.4.3	Design	222
6.4.4	Implementation (Coding)	224
6.5	Managing a Program's Namespace	227
6.5.1	Module Variables, Parameters, and Temporary Variables	227
6.5.2	Scope	228
6.5.3	Lifetime	229
6.5.4	Default (Keyword) Arguments	230
6.5	Exercises	232
6.6	Higher-Order Functions (Advanced Topic)	233
6.6.1	Functions as First-Class Data Objects	233
6.6.2	Mapping	234
6.6.3	Filtering	236
6.6.4	Reducing	237
6.6.5	Using lambda to Create Anonymous Functions	237
6.6.6	Creating Jump Tables	238

6.6	Exercises.....	239
	Summary.....	240
	Review Questions.....	242
	Projects.....	244
[CHAPTER] 7	SIMPLE GRAPHICS AND IMAGE PROCESSING	247
7.1	Simple Graphics.....	248
7.1.1	Overview of Turtle Graphics.....	248
7.1.2	Turtle Operations.....	249
7.1.3	Object Instantiation and the <code>turtle</code> Module.....	252
7.1.4	Drawing Two-Dimensional Shapes.....	254
7.1.5	Taking a Random Walk.....	255
7.1.6	Colors and the RGB System.....	256
7.1.7	Example: Drawing with Random Colors.....	257
7.1.8	Examining an Object's Attributes.....	259
7.1.9	Manipulating a Turtle's Screen.....	259
7.1.10	Setting up a <code>cfg</code> File and Running IDLE.....	260
7.1	Exercises.....	261
7.2	Case Study: Recursive Patterns in Fractals.....	262
7.2.1	Request.....	263
7.2.2	Analysis.....	263
7.2.3	Design.....	264
7.2.4	Implementation (Coding).....	266
7.3	Image Processing.....	267
7.3.1	Analog and Digital Information.....	267
7.3.2	Sampling and Digitizing Images.....	268
7.3.3	Image File Formats.....	268
7.3.4	Image-Manipulation Operations.....	269
7.3.5	The Properties of Images.....	270
7.3.6	The <code>images</code> Module.....	270
7.3.7	A Loop Pattern for Traversing a Grid.....	274
7.3.8	A Word on Tuples.....	275
7.3.9	Converting an Image to Black and White.....	276
7.3.10	Converting an Image to Grayscale.....	278
7.3.11	Copying an Image.....	279
7.3.12	Blurring an Image.....	280
7.3.13	Edge Detection.....	281
7.3.14	Reducing the Image Size.....	282
7.3	Exercises.....	284
	Summary.....	285
	Review Questions.....	286
	Projects.....	288
[CHAPTER] 8	DESIGN WITH CLASSES	293
8.1	Getting Inside Objects and Classes.....	294
8.1.1	A First Example: The <code>Student</code> Class.....	295
8.1.2	Docstrings.....	298
8.1.3	Method Definitions.....	298

	8.1.4	The <code>__init__</code> Method and Instance Variables.....	299
	8.1.5	The <code>__str__</code> Method.....	300
	8.1.6	Accessors and Mutators	300
	8.1.7	The Lifetime of Objects	301
	8.1.8	Rules of Thumb for Defining a Simple Class.....	302
		Exercises.....	303
8.1		Case Study: Playing the Game of Craps	303
8.2	8.2.1	Request	303
	8.2.2	Analysis.....	303
	8.2.3	Design.....	304
	8.2.4	Implementation (Coding).....	306
8.3		Data-Modeling Examples.....	309
	8.3.1	Rational Numbers.....	309
	8.3.2	Rational Number Arithmetic and Operator Overloading.....	311
	8.3.3	Comparison Methods.....	312
	8.3.4	Equality and the <code>__eq__</code> Method.....	314
	8.3.5	Savings Accounts and Class Variables	315
	8.3.6	Putting the Accounts into a Bank.....	317
	8.3.7	Using <code>pickle</code> for Permanent Storage of Objects.....	319
	8.3.8	Input of Objects and the <code>try-except</code> Statement.....	320
	8.3.9	Playing Cards	321
8.3		Exercises.....	325
8.4		Case Study: An ATM.....	325
	8.4.1	Request	325
	8.4.2	Analysis.....	325
	8.4.3	Design.....	327
	8.4.4	Implementation (Coding).....	329
8.5		Structuring Classes with Inheritance and Polymorphism.....	331
	8.5.1	Inheritance Hierarchies and Modeling	332
	8.5.2	Example: A Restricted Savings Account.....	333
	8.5.3	Example: The Dealer and a Player in the Game of Blackjack	335
	8.5.4	Polymorphic Methods.....	340
	8.5.5	Abstract Classes	340
	8.5.6	The Costs and Benefits of Object-Oriented Programming.....	341
8.5		Exercises.....	343
		Summary.....	343
		Review Questions	345
		Projects.....	346

[CHAPTER] 9 GRAPHICAL USER INTERFACES 349

9.1		The Behavior of Terminal-Based Programs and GUI-Based Programs.....	350
	9.1.1	The Terminal-Based Version.....	350
	9.1.2	The GUI-Based Version.....	351
	9.1.3	Event-Driven Programming.....	353
9.1		Exercises.....	355
9.2		Coding Simple GUI-Based Programs	355
	9.2.1	Windows and Labels.....	356
	9.2.2	Displaying Images.....	357
	9.2.3	Command Buttons and Responding to Events.....	358
	9.2.4	Viewing the Images of Playing Cards	360

	9.2.5	Entry Fields for the Input and Output of Text	363
	9.2.6	Using Pop-up Dialog Boxes	365
9.2		Exercises.....	366
9.3		Case Study: A GUI-Based ATM.....	367
	9.3.1	Request	367
	9.3.2	Analysis	367
	9.3.3	Design.....	368
	9.3.4	Implementation (Coding)	369
9.4		Other Useful GUI Resources	372
	9.4.1	Colors	373
	9.4.2	Text Attributes.....	373
	9.4.3	Sizing and Justifying an Entry	374
	9.4.4	Sizing the Main Window.....	375
	9.4.5	Grid Attributes	376
	9.4.6	Using Nested Frames to Organize Components.....	380
	9.4.7	Multi-Line Text Widgets.....	381
	9.4.8	Scrolling List Boxes	384
	9.4.9	Mouse Events	387
	9.4.10	Keyboard Events	388
9.4		Exercises.....	389
		Summary	390
		Review Questions	391
		Projects.....	392

[CHAPTER] 10

MULTITHREADING, NETWORKS, AND CLIENT/SERVER PROGRAMMING 395

10.1		Threads and Processes	396
	10.1.1	Threads.....	397
	10.1.2	Sleeping Threads.....	400
	10.1.3	Producer, Consumer, and Synchronization	402
10.1		Exercises.....	409
10.2		Networks, Clients, and Servers.....	409
	10.2.1	IP Addresses	409
	10.2.2	Ports, Servers, and Clients.....	411
	10.2.3	Sockets and a Day/Time Client Script.....	412
	10.2.4	A Day/Time Server Script.....	414
	10.2.5	A Two-Way Chat Script.....	416
	10.2.6	Handling Multiple Clients Concurrently	418
	10.2.7	Setting Up Conversations for Others	420
10.2		Exercises.....	422
10.3		Case Study: A Multi-Client Chat Room	423
	10.3.1	Request	423
	10.3.2	Analysis	423
	10.3.3	Design.....	424
	10.3.4	Implementation (Coding)	425
		Summary	427
		Review Questions	428
		Projects.....	430

[ONLINE CHAPTER] **11**

SEARCHING, SORTING, AND COMPLEXITY ANALYSIS

(Available on the publisher's Web site at www.cengagebrain.com.)

11.1	Measuring the Efficiency of Algorithms
11.1.1	Measuring the Run Time of an Algorithm
11.1.2	Counting Instructions
11.1.3	Measuring the Memory Used by an Algorithm
11.1	Exercises
11.2	Complexity Analysis
11.2.1	Orders of Complexity
11.2.2	Big-O Notation
11.2.3	The Role of the Constant of Proportionality
11.2	Exercises
11.3	Search Algorithms
11.3.1	Search for a Minimum
11.3.2	Linear Search of a List
11.3.3	Best-Case, Worst-Case, and Average-Case Performance
11.3.4	Binary Search of a List
11.3.5	Comparing Data Items
11.3	Exercises
11.4	Sort Algorithms
11.4.1	Selection Sort
11.4.2	Bubble Sort
11.4.3	Insertion Sort
11.4.4	Best-Case, Worst-Case, and Average-Case Performance Revisited
11.4	Exercises
11.5	An Exponential Algorithm: Recursive Fibonacci
11.6	Converting Fibonacci to a Linear Algorithm
11.7	Case Study: An Algorithm Profiler
11.7.1	Request
11.7.2	Analysis
11.7.3	Design
11.7.4	Implementation (Coding)
	Summary
	Review Questions
	Projects

[APPENDIX] A	PYTHON RESOURCES	433
A.1	Installing Python on Your Computer	434
A.2	Using the Terminal Command Prompt, IDLE, and Other IDEs.....	434
[APPENDIX] B	INSTALLING THE images LIBRARY	437
[APPENDIX] C	API FOR IMAGE PROCESSING	439
[APPENDIX] D	TRANSITION FROM PYTHON TO JAVA AND C++	441
GLOSSARY		443
INDEX		455

PREFACE

Welcome to *Fundamentals of Python: First Programs*. This text is intended for a course in programming and problem-solving. It covers the material taught in a typical Computer Science 1 course (CS1) at the undergraduate level.

This book covers five major aspects of computing:

- 1 **Programming Basics**—Data types, control structures, algorithm development, and program design with functions are basic ideas that you need to master in order to solve problems with computers. This book examines these core topics in detail and gives you practice employing your understanding of them to solve a wide range of problems.
- 2 **Object-Oriented Programming (OOP)**—Object-Oriented Programming is the dominant programming paradigm used to develop large software systems. This book introduces you to the fundamental principles of OOP and enables you to apply them successfully.
- 3 **Data and Information Processing**—Most useful programs rely on data structures to solve problems. These data structures include strings, arrays, files, lists, and dictionaries. This book introduces you to these commonly used data structures, with examples that illustrate criteria for selecting the appropriate data structures for given problems.
- 4 **Software Development Life Cycle**—Rather than isolate software development techniques in one or two chapters, this book deals with them throughout in the context of numerous case studies. Among other things, you'll learn that coding a program is often not the most difficult or challenging aspect of problem solving and software development.
- 5 **Contemporary Applications of Computing**—The best way to learn about programming and problem solving is to create interesting programs with real-world applications. In this book, you'll begin by creating applications that involve numerical problems and text processing. For example, you'll learn the basics of encryption techniques such as those that are used to make your credit card number and other information secure on the Internet. But unlike many other introductory texts, this one does not restrict itself to problems involving numbers and text. Most contemporary applications involve graphical user interfaces, event-driven programming, graphics, and network communications. These topics are presented in optional, standalone chapters.

Why Python?

Computer technology and applications have become increasingly more sophisticated over the past two decades, and so has the computer science curriculum, especially at the introductory level. Today's students learn a bit of programming and problem-solving, and are then expected to move quickly into topics like software development, complexity analysis, and data structures that, twenty years ago, were relegated to advanced courses. In addition, the ascent of object-oriented programming as the dominant paradigm of problem solving has led instructors and textbook authors to bring powerful, industrial-strength programming languages such as C++ and Java into the introductory curriculum. As a result, instead of experiencing the rewards and excitement of solving problems with computers, beginning computer science students often become overwhelmed by the combined tasks of mastering advanced concepts as well as the syntax of a programming language.

This book uses the Python programming language as a way of making the first year of computer science more manageable and attractive for students and instructors alike. Python has the following pedagogical benefits:

- Python has simple, conventional syntax. Python statements are very close to those of pseudocode algorithms, and Python expressions use the conventional notation found in algebra. Thus, students can spend less time learning the syntax of a programming language and more time learning to solve interesting problems.
- Python has safe semantics. Any expression or statement whose meaning violates the definition of the language produces an error message.
- Python scales well. It is very easy for beginners to write simple programs in Python. Python also includes all of the advanced features of a modern programming language, such as support for data structures and object-oriented software development, for use when they become necessary.
- Python is highly interactive. Expressions and statements can be entered at an interpreter's prompts to allow the programmer to try out experimental code and receive immediate feedback. Longer code segments can then be composed and saved in script files to be loaded and run as modules or standalone applications.
- Python is general purpose. In today's context, this means that the language includes resources for contemporary applications, including media computing and networks.
- Python is free and is in widespread use in industry. Students can download Python to run on a variety of devices. There is a large Python user community, and expertise in Python programming has great resume value.

To summarize these benefits, Python is a comfortable and flexible vehicle for expressing ideas about computation, both for beginners and for experts as well. If students learn these ideas well in the first course, they should have no problems making a quick transition to other languages needed for courses later in the curriculum. Most importantly, beginning students will spend less time staring at a computer screen and more time thinking about interesting problems to solve.

Organization of the Book

The approach of this text is easygoing, with each new concept introduced only when it is needed.

Chapter 1 introduces computer science by focusing on two fundamental ideas, algorithms and information processing. A brief overview of computer hardware and software, followed by an extended discussion of the history of computing, sets the context for computational problem solving.

Chapters 2 and 3 cover the basics of problem solving and algorithm development using the standard control structures of expression evaluation, sequencing, Boolean logic, selection, and iteration with the basic numeric data types. Emphasis in these chapters is on problem solving that is both systematic and experimental, involving algorithm design, testing, and documentation.

Chapters 4 and 5 introduce the use of the strings, text files, lists, and dictionaries. These data structures are both remarkably easy to manipulate in Python and support some interesting applications. Chapter 5 also introduces simple function definitions as a way of organizing algorithmic code.

Chapter 6 explores the technique and benefits of procedural abstraction with function definitions. Top-down design, stepwise refinement, and recursive design with functions are examined as means of structuring code to solve complex problems. Details of namespace organization (parameters, temporary variables, and module variables) and communication among software components are discussed. An optional section on functional programming with higher-order functions shows how to exploit functional design patterns to simplify solutions.

Chapter 7 focuses on the use of existing objects and classes to compose programs. Special attention is paid to the interface, or set of methods, of a class of objects and the manner in which objects cooperate to solve problems. This chapter also introduces two contemporary applications of computing, graphics and image processing—areas in which object-based programming is particularly useful.

Chapter 8 introduces object-oriented design with class and method definitions. Several examples of simple class definitions from different application domains are presented. Some of these are then integrated into more realistic applications, to show how object-oriented software components can be used to build complex systems. Emphasis is on designing appropriate interfaces for classes that exploit inheritance and polymorphism.

Chapters 9 and 10 cover advanced material related to two important areas of computing: graphical user interfaces and networks. Although these two chapters are entirely optional, they give students challenging experiences at the end of the first course. Chapter 9 contrasts the event-driven model of GUI programs with the process-driven model of terminal-based programs. The creation and layout of GUI components are explored, as well as the decomposition of a GUI-based program using the model/view/controller pattern. Chapter 10 introduces multi-threaded programs and the construction of simple network-based client/server applications.

Chapter 11 covers some topics addressed at the beginning of a traditional CS2 course, and is available on the publisher's Web site. This chapter introduces complexity analysis with big-O notation. Enough material is presented to enable you to perform simple analyses of the running time and memory usage of algorithms and data structures, using search and sort algorithms as examples.

Special Features

This book explains and develops concepts carefully, using frequent examples and diagrams. New concepts are then applied in complete programs to show how they aid in solving problems. The chapters place an early and consistent emphasis on good writing habits and neat, readable documentation.

The book includes several other important features:

- **Case studies**—These present complete Python programs ranging from the simple to the substantial. To emphasize the importance and usefulness of the software development life cycle, case studies are discussed in the framework of a user request, followed by analysis, design, implementation, and suggestions for testing, with well-defined tasks performed at each stage. Some case studies are extended in end-of-chapter programming projects.
- **Chapter objectives and chapter summaries**—Each chapter begins with a set of learning objectives and ends with a summary of the major concepts covered in the chapter.
- **Key terms and a glossary**—When a technical term is introduced in the text, it appears in boldface. Definitions of the key terms are also collected in a glossary.

- Exercises—Most major sections of each chapter end with exercise questions that reinforce the reading by asking basic questions about the material in the section. Each chapter ends with a set of review exercises.
- Programming projects—Each chapter ends with a set of programming projects of varying difficulty.
- A software toolkit for image processing—This book comes with an open-source Python toolkit for the easy image processing discussed in Chapter 7. The toolkit can be obtained from the student downloads page on www.course.com, or at <http://home.wlu.edu/~lambertk/python/>
- Appendices—Three appendices include information on obtaining Python resources, installing the toolkit, and using the toolkit's interface.

Supplemental Resources

The following supplemental materials are available when this book is used in a classroom setting. All of the teaching tools available with this book are provided to the instructor on a single CD-ROM.

Electronic Instructor's Manual

The Instructor's Manual that accompanies this textbook includes:

- Additional instructional material to assist in class preparation, including suggestions for lecture topics.
- Solutions to all the end-of-chapter materials, including the Programming Exercises.

ExamView®

This textbook is accompanied by ExamView, a powerful testing software package that allows instructors to create and administer printed, computer (LAN-based), and Internet exams. ExamView includes hundreds of questions that correspond to the topics covered in this text, enabling students to generate detailed study guides that include page references for further review. These computer-based and Internet testing components allow students to take exams at their computers, and save the instructor time because each exam is graded automatically.

PowerPoint Presentations

This book comes with Microsoft PowerPoint slides for each chapter. These are included as a teaching aid either to make available to students on the network for chapter review, or to be used during classroom presentations. Instructors can modify slides or add their own slides to tailor their presentations.

Distance Learning

Course Technology is proud to offer online courses in WebCT and Blackboard. For more information on how to bring distance learning to your course, contact your local Cengage Learning sales representative.

Source Code

The source code is available at www.cengagebrain.com—and also is available on the Instructor Resources CD-ROM. If an input file is needed to run a program, it is included with the source code.

Solution files

The solution files for all programming exercises are available at www.cengagebrain.com and are available on the Instructor Resources CD-ROM. If an input file is needed to run a programming exercise, it is included with the solution file.

We Appreciate Your Feedback

We have tried to produce a high-quality text, but should you encounter any errors, please report them to lambertk@wlu.edu or computerscience@cengage.com. A list of errata, should they be found, as well as other information about the book, will be posted on the Web site <http://home.wlu.edu/~lambertk/python/> and with the student resources at www.cengagebrain.com.

Acknowledgments

I would like to thank my contributing author, Martin Osborne, for many years of advice, friendly criticism, and encouragement on several of my book projects. To my colleague, Joshua Stough, and our students at Washington and Lee University for classroom testing this book over several semesters.

In addition, I would like to thank the following reviewers for the time and effort they contributed to *Fundamentals of Python*: Paul Albee, Central Michigan University; Andrew Danner, Swarthmore College; Susan Fox, Macalester College; Robert Franks, Central College; and Jim Slack, Minnesota State University, Mankato. Also, thank you to the following reviewers who contributed their thoughts on the original book proposal: Christian Blouin, Dalhousie University; Margaret Iwobi, Binghamton University; Sam Midkiff, Purdue University; and Ray Morehead, West Virginia University.

Also, thank you to the individuals at Course Technology who helped to assure that the content of all data and solution files used for this text were correct and accurate: Chris Scriver, MQA Project Leader and Serge Palladino, MQA Tester.

Finally, thanks to several other people whose work made this book possible: Ann Shaffer, Developmental Editor; Brandi Shailer, Acquisitions Editor, Course Technology; Alyssa Pratt, Senior Product Manager, Course Technology; and Jennifer Feltri, Content Project Manager, Course Technology.

Dedication

To my students in Computer Science 111

Kenneth A. Lambert

Lexington, VA

[CHAPTER] 1 Introduction

After completing this chapter, you will be able to

- Describe the basic features of an algorithm
- Explain how hardware and software collaborate in a computer's architecture
- Give a brief history of computing
- Compose and run a simple Python program

As a reader of this book, you almost certainly have played a video game and listened to music on a CD player. It's likely that you have watched a movie on a DVD player and prepared a snack in a microwave oven. Chances are that you have made at least one phone call to or from a cell phone. You and your friends have most likely used a desktop computer or a laptop computer, not to mention digital cameras and handheld music and video players.

All of these devices have something in common: they are or contain computers. Computer technology makes them what they are. Devices that rely on computer technology are almost everywhere, not only in our homes, but also in our schools, where we work, and where we play. Computer technology plays an important role in entertainment, education, medicine, manufacturing, communications, government, and commerce. It has been said that we have digital lifestyles and that we live in an information age with an information-based economy. Some people even claim that nature itself performs computations on information structures present in DNA and in the relationships among subatomic particles.

It's difficult to imagine our world without computers, although we don't think about the actual computers very much. It's also hard to imagine that the human race did without computer technology

for thousands of years, and that the world as we know it has been so involved in and with computer technology for only the past 25 years or so.

In the chapters that follow, you will learn about computer science, which is the study of computation that has made this new technology and this new world possible. You will also learn how to use computers effectively and appropriately to enhance your own life and the lives of others.

1.1

Two Fundamental Ideas of Computer Science: Algorithms and Information Processing

Like most areas of study, computer science focuses on a broad set of interrelated ideas. Two of the most basic ones are **algorithms** and **information processing**. In this section, these ideas are introduced in an informal way. We will examine them in more detail in later chapters.

1.1.1

Algorithms

People computed long before the invention of modern computing devices, and many continue to use computing devices that we might consider primitive. For example, consider how merchants made change for customers in marketplaces before the existence of credit cards, pocket calculators, or cash registers. Making change can be a complex activity. It probably took you some time to learn how to do it, and it takes some mental effort to get it right every time. Let's consider what's involved in this process.

The first step is to compute the difference between the purchase price and the amount of money that the customer gives the merchant. The result of this calculation is the total amount that the merchant must return to the purchaser. For example, if you buy a dozen eggs at the farmers' market for \$2.39 and you give the farmer a \$10 bill, she should return \$7.61 to you. To produce this amount, the merchant selects the appropriate coins and bills that, when added to \$2.39, make \$10.00.

Few people can subtract three-digit numbers without resorting to some manual aids, such as pencil and paper. As you learned in grade school, you can carry out subtraction with pencil and paper by following a sequence of well-defined steps. You have probably done this many times but never made a list of the

specific steps involved. Making such lists to solve problems is something computer scientists do all the time. For example, the following list of steps describes the process of subtracting two numbers using a pencil and paper:

- Step 1** Write down the two numbers, with the larger number above the smaller number and their digits aligned in columns from the right.
- Step 2** Assume that you will start with the rightmost column of digits and work your way left through the various columns.
- Step 3** Write down the difference between the two digits in the current column of digits, borrowing a 1 from the top number's next column to the left if necessary.
- Step 4** If there is no next column to the left, stop. Otherwise, move to the next column to the left, and go to Step 3.

If the **computing agent** (in this case a human being) follows each of these simple steps correctly, the entire process results in a correct solution to the given problem. We assume in Step 3 that the agent already knows how to compute the difference between the two digits in any given column, borrowing if necessary.

To make change, most people can select the combination of coins and bills that represent the correct change amount without any manual aids, other than the coins and bills. But the mental calculations involved can still be described in a manner similar to the preceding steps, and we can resort to writing them down on paper if there is a dispute about the correctness of the change.

The sequence of steps that describes each of these computational processes is called an **algorithm**. Informally, an algorithm is like a recipe. It provides a set of instructions that tells us how to do something, such as make change, bake bread, or put together a piece of furniture. More precisely, an algorithm describes a process that ends with a solution to a problem. The algorithm is also one of the fundamental ideas of computer science. An algorithm has the following features:

- 1 An algorithm consists of a finite number of instructions.
- 2 Each individual instruction in an algorithm is well defined. This means that the action described by the instruction can be performed effectively or be **executed** by a computing agent. For example, any computing agent capable of arithmetic can compute the difference between two digits. So an algorithmic step that says “compute the difference between two digits” would be well defined. On the other hand, a step that says “divide a number by 0” is not well defined, because no computing agent could carry it out.

- 3 An algorithm describes a process that eventually halts after arriving at a solution to a problem. For example, the process of subtraction halts after the computing agent writes down the difference between the two digits in the leftmost column of digits.
- 4 An algorithm solves a general class of problems. For example, an algorithm that describes how to make change should work for any two amounts of money whose difference is greater than or equal to \$0.00.

Creating a list of steps that describe how to make change might not seem like a major accomplishment to you. But the ability to break a task down into its component parts is one of the main jobs of a computer programmer. Once we write an algorithm to describe a particular type of computation, a machine can be built to do the computing. Put another way, if we can develop an algorithm to solve a problem, we can automate the task of solving the problem. You might not feel compelled to write a computer program to automate the task of making change, because you can probably already make change yourself fairly easily. But suppose you needed to do a more complicated task—such as sorting a list of 100 names. In that case, a computer program would be very handy.

Computers can be designed to run a small set of algorithms for performing specialized tasks such as operating a microwave oven. But we can also build computers, like the one on your desktop, that are capable of performing a task described by any algorithm. These computers are truly general-purpose problem-solving machines. They are unlike any machines we have ever built before, and they have formed the basis of the completely new world in which we live.

Later in this book, we introduce a notation for expressing algorithms and some suggestions for designing algorithms. You will see that algorithms and algorithmic thinking are critical underpinnings of any computer system.

1.1.2 Information Processing

Since human beings first learned to write several thousand years ago, they have processed information. Information itself has taken many forms in its history, from the marks impressed on clay tablets in ancient Mesopotamia, to the first written texts in ancient Greece, to the printed words in the books, newspapers, and magazines mass-produced since the European Renaissance, to the abstract symbols of modern mathematics and science used during the past 350 years. Only recently, however, have human beings developed the capacity to automate the processing of information by building computers. In the modern world of computers, information is also commonly referred to as **data**. But what is information?

Like mathematical calculations, information processing can be described with algorithms. In our earlier example of making change, the subtraction steps involved manipulating symbols used to represent numbers and money. In carrying out the instructions of any algorithm, a computing agent manipulates information. The computing agent starts with some given information (known as **input**), transforms this information according to well-defined rules, and produces new information, known as **output**.

It is important to recognize that the algorithms that describe information processing can also be represented as information. Computer scientists have been able to represent algorithms in a form that can be executed effectively and efficiently by machines. They have also designed real machines, called electronic digital computers, which are capable of executing algorithms.

Computer scientists more recently discovered how to represent many other things, such as images, music, human speech, and video, as information. Many of the media and communication devices that we now take for granted would be impossible without this new kind of information processing. We examine many of these achievements in more detail in later chapters.

1.1 Exercises

These short end-of-section exercises are intended to stimulate your thinking about computing.

- 1 List three common types of computing agents.
- 2 Write an algorithm that describes the second part of the process of making change (counting out the coins and bills).
- 3 Write an algorithm that describes a common task, such as baking a cake or operating a DVD player.
- 4 Describe an instruction that is not well defined and thus could not be included as a step in an algorithm. Give an example of such an instruction.
- 5 In what sense is a desktop computer a general-purpose problem-solving machine?
- 6 List four devices that use computers and describe the information that they process. (*Hint*: Think of the inputs and outputs of the devices.)

1.2

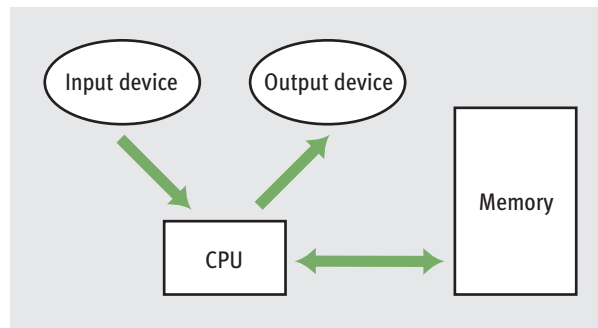
The Structure of a Modern Computer System

We now give a brief overview of the structure of modern computer systems. A modern computer system consists of **hardware** and **software**. Hardware consists of the physical devices required to execute algorithms. Software is the set of these algorithms, represented as **programs** in particular **programming languages**. In the discussion that follows, we focus on the hardware and software found in a typical desktop computer system, although similar components are also found in other computer systems, such as handheld devices and ATMs (automatic teller machines).

1.2.1

Computer Hardware

The basic hardware components of a computer are **memory**, a **central processing unit (CPU)**, and a set of **input/output devices**, as shown in Figure 1.1.



[FIGURE 1.1] Hardware components of a modern computer system

Human users primarily interact with the input and output devices. The input devices include a keyboard, a mouse, and a microphone. Common output devices include a monitor and speakers. Computers can also communicate with the external world through various **ports** that connect them to **networks** and to other devices such as handheld music players and digital cameras. The purpose of most of the input devices is to convert information that human beings deal with, such as text, images, and sounds, into information for computational processing. The purpose of most output devices is to convert the results of this processing back to human-usable form.

Computer memory is set up to represent and store information in electronic form. Specifically, information is stored as patterns of **binary digits** (1s and 0s). To understand how this works, consider a basic device such as a light switch, which can only be in one of two states, on or off. Now suppose there is a bank of switches that control 16 small lights in a row. By turning the switches off or on, we can represent any pattern of 16 binary digits (1s and 0s) as patterns of lights that are on or off. As we will see later in this book, computer scientists have discovered how to represent any information, including text, images, and sound, in binary form.

Now, suppose there are 8 of these groups of 16 lights. We can select any group of lights and examine or change the state of each light within that collection. We have just developed a tiny model of computer memory. This memory has 8 cells, each of which can store 16 **bits** of binary information. A diagram of this model, in which the memory cells are filled with binary digits, is shown in Figure 1.2. This memory is also sometimes called **primary** or **internal** or **random access memory (RAM)**.

Cell 7	1	1	0	1	1	1	1	0	1	1	1	1	1	1	0	1
Cell 6	1	0	1	1	0	1	1	1	1	1	0	1	1	1	1	1
Cell 5	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1
Cell 4	1	0	1	1	1	0	1	1	1	1	1	0	1	1	1	1
Cell 3	1	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1
Cell 2	0	0	1	1	1	1	0	1	1	1	0	1	1	1	0	1
Cell 1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	1	1
Cell 0	1	1	1	0	1	1	0	1	1	1	1	1	1	1	1	0

[FIGURE 1.2] A model of computer memory

The information stored in memory can represent any type of data, such as numbers, text, images, or sound, or the instructions of a program. We can also store in memory an algorithm encoded as binary instructions for the computer. Once the information is stored in memory, we typically want to do something with it—that is, we want to process it. The part of a computer that is responsible for processing data is the central processing unit (CPU). This device, which is also sometimes called a **processor**, consists of electronic switches arranged to perform simple logical, arithmetic, and control operations. The CPU executes an algorithm by fetching its binary instructions from memory, decoding them, and executing them. Executing an instruction might involve fetching other binary information—the data—from memory as well.

The processor can locate data in a computer's primary memory very quickly. However, these data exist only as long as electric power comes into the computer. If the power fails or is turned off, the data in primary memory are lost. Clearly, a more permanent type of memory is needed to preserve data. This more permanent type of memory is called **external** or **secondary memory**, and it comes in several forms. **Magnetic storage media**, such as tapes and hard disks, allow bit patterns to be stored as patterns on a magnetic field. **Semiconductor storage media**, such as flash memory sticks, perform much the same function with a different technology, as do **optical storage media**, such as CDs and DVDs. Some of these secondary storage media can hold much larger quantities of information than the internal memory of a computer.

1.2.2

Computer Software

You have learned that a computer is a general-purpose, problem-solving machine. To solve any computable problem, a computer must be capable of executing any algorithm. Because it is impossible to anticipate all of the problems for which there are algorithmic solutions, there is no way to “hardwire” all potential algorithms into a computer's hardware. Instead, we build some basic operations into the hardware's processor and require any algorithm to use them. The algorithms are converted to binary form and then loaded, with their data, into the computer's memory. The processor can then execute the algorithms' instructions by running the hardware's more basic operations.

Any programs that are stored in memory so that they can be executed later are called software. A program stored in computer memory must be represented in binary digits, which is also known as **machine code**. Loading machine code into computer memory one digit at a time would be a tedious, error-prone task for human beings. It would be convenient if we could automate this process to get it right every time. For this reason, computer scientists have developed another program, called a **loader**, to perform this task. A loader takes a set of machine language instructions as input and loads them into the appropriate memory locations. When the loader is finished, the machine language program is ready to execute. Obviously, the loader cannot load itself into memory, so this is one of those algorithms that must be hardwired into the computer.

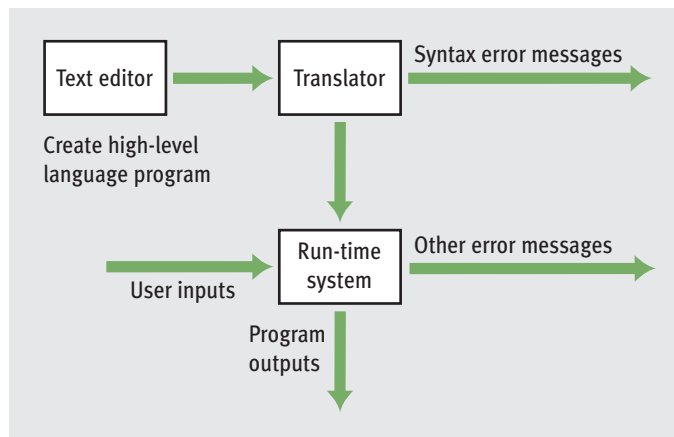
Now that a loader exists, we can load and execute other programs that make the development, execution, and management of programs easier. This type of software is called **system software**. The most important example of system software is a computer's **operating system**. You are probably already familiar with at least one of the most popular operating systems, such as Linux, Apple's Mac OS,

and Microsoft Windows. An operating system is responsible for managing and scheduling several concurrently running programs. It also manages the computer's memory, including the external storage, and manages communications between the CPU, the input/output devices, and other computers on a network. An important part of any operating system is its **file system**, which allows human users to organize their data and programs in permanent storage. Another important function of an operating system is to provide **user interfaces**—that is, ways for the human user to interact with the computer's software. A **terminal-based interface** accepts inputs from a keyboard and displays text output on a monitor screen. A modern **graphical user interface (GUI)** organizes the monitor screen around the metaphor of a desktop, with windows containing icons for folders, files, and applications. This type of user interface also allows the user to manipulate images with a pointing device such as a mouse.

Another major type of software is called **applications software**, or simply **applications**. An application is a program that is designed for a specific task, such as editing a document or displaying a Web page. Applications include Web browsers, word processors, spreadsheets, database managers, graphic design packages, music production systems, and games, among many others. As you begin to learn to write computer programs, you will focus on writing simple applications.

As you have learned, computer hardware can execute only instructions that are written in binary form—that is, in machine language. Writing a machine language program, however, would be an extremely tedious, error-prone task. To ease the process of writing computer programs, computer scientists have developed **high-level programming languages** for expressing algorithms. These languages resemble English and allow the author to express algorithms in a form that other people can understand.

A programmer typically starts by writing high-level language statements in a **text editor**. The programmer then runs another program called a **translator** to convert the high-level program code into executable code. Because it is possible for a programmer to make grammatical mistakes even when writing high-level code, the translator checks for **syntax errors** before it completes the translation process. If it detects any of these errors, the translator alerts the programmer via error messages. The programmer then has to revise the program. If the translation process succeeds without a syntax error, the program can be executed by the **run-time system**. The run-time system might execute the program directly on the hardware or run yet another program called an **interpreter** or **virtual machine** to execute the program. Figure 1.3 shows the steps and software used in the coding process.



[FIGURE 1.3] Software used in the coding process

1.2 Exercises

- 1 List two examples of input devices and two examples of output devices.
- 2 What does the central processing unit (CPU) do?
- 3 How is information represented in hardware memory?
- 4 What is the difference between a terminal-based interface and a graphical user interface?
- 5 What role do translators play in the programming process?

1.3 A Not-So-Brief History of Computing Systems

Now that we have in mind some of the basic ideas of computing and computer systems, let's take a moment to examine how they have taken shape in history. Figure 1.4 summarizes some of the major developments in the history of computing. The discussion that follows provides more details about these developments.

Approximate Dates	Major Developments
Before 1800	• Mathematicians develop and use algorithms • Abacus used as a calculating aide • First mechanical calculators built by Pascal and Leibniz
1800–1930	• Jacquard’s loom • Babbage’s Analytical Engine • Boole’s system of logic • Hollerith’s punch card machine
1930s	• Turing publishes results on computability • Shannon’s theory of information and digital switching
1940s	• First electronic digital computers
1950s	• First symbolic programming languages • Transistors make computers smaller, faster, more durable, less expensive • Emergence of data-processing applications
1960–1975	• Integrated circuits accelerate the miniaturization of hardware • First minicomputers • Time-sharing operating systems • Interactive user interfaces with keyboards and monitors • Proliferation of high-level programming languages • Emergence of a software industry and the academic study of computer science and computer engineering
1975–1990	• First microcomputers and mass-produced personal computers • Graphical user interfaces become widespread • Networks and the Internet
1990s	• Optical storage for multimedia applications, images, sound, and video • World Wide Web and e-commerce • Laptop computers
2000–present	• Embedded computing • Wireless computing • Computers used in enormous variety of cars, household appliances, and industrial equipment

[FIGURE 1.4] Summary of major developments in the history of computing

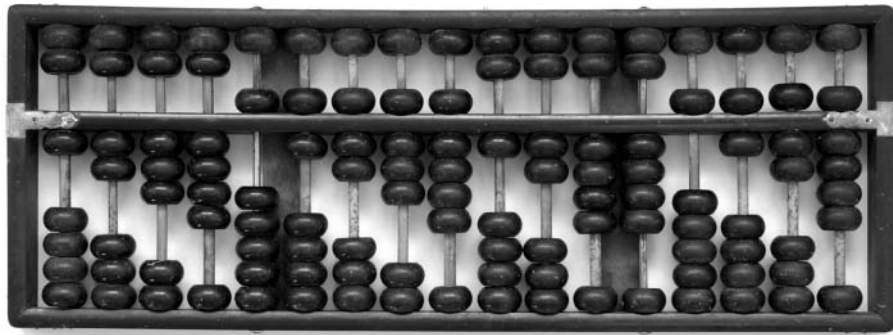
1.3.1

Before Electronic Digital Computers

Ancient mathematicians developed the first algorithms. The word “algorithm” comes from the name of a Persian mathematician, Muhammad ibn Musa Al-Khawarizmi, who wrote several mathematics textbooks in the ninth century.

About 2,300 years ago, the Greek mathematician Euclid, the inventor of geometry, developed an algorithm for computing the greatest common divisor of two numbers.

A device known as the **abacus** also appeared in ancient times. The abacus helped people perform simple arithmetic. Users calculated sums and differences by sliding beads on a grid of wires (see Figure 1.5a). The configuration of beads on the abacus served as the data.



[a] Abacus Image © Lim ChewHow, 2008. Used under license from Shutterstock.com.

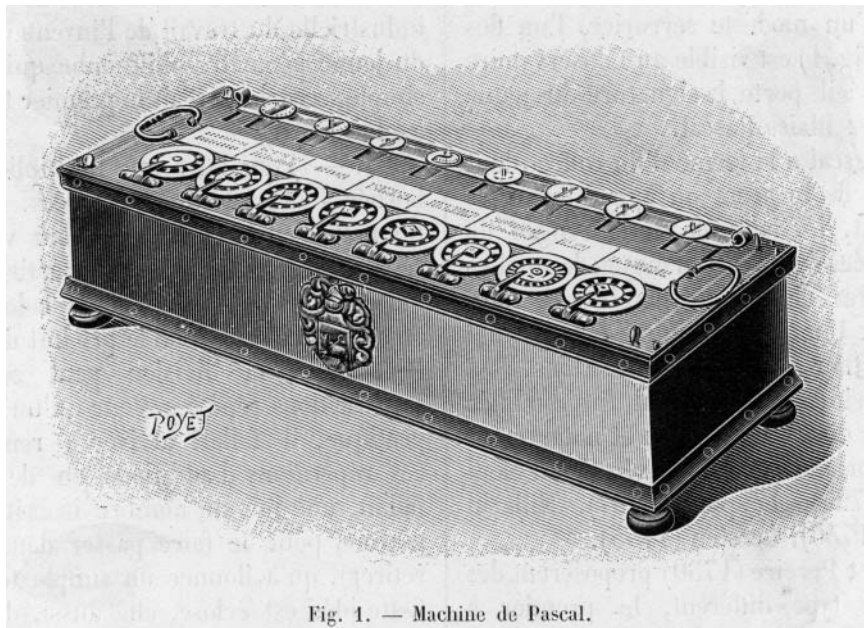
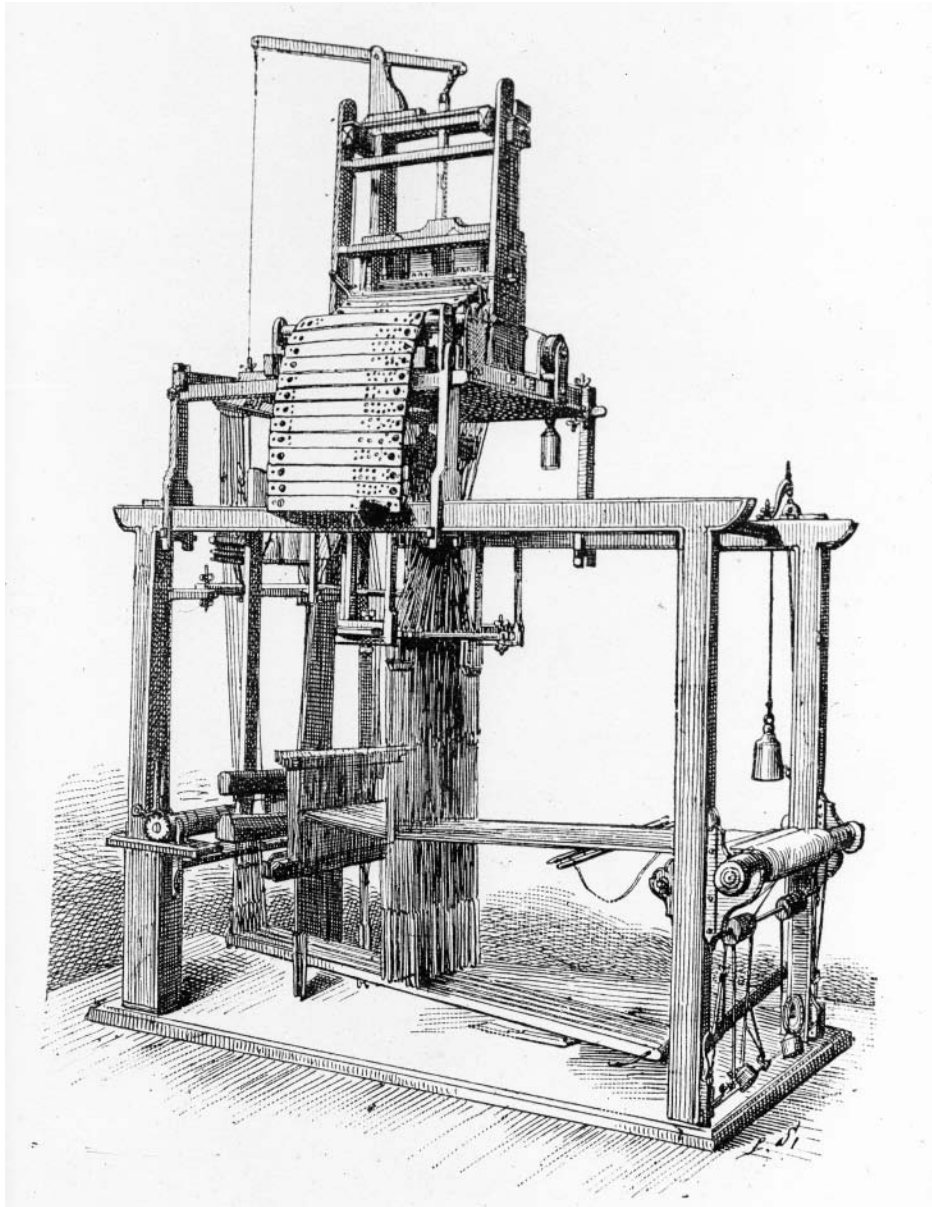


Fig. 1. — Machine de Pascal.

[b] Pascal's Calculator Image © Mary Evans/Photo Reasearchers, Inc.



[c] Jacquard's Loom Image © Roger Viollet/Getty Images

[FIGURE 1.5] Some early computing devices

In the seventeenth century, the French mathematician Blaise Pascal (1623–1662) built one of the first mechanical devices to automate the process of

addition (see Figure 1.5b). The addition operation was embedded in the configuration of gears within the machine. The user entered the two numbers to be added by rotating some wheels. The sum or output number appeared on another rotating wheel. The German mathematician Gottfried Leibnitz (1646–1716) built another mechanical calculator that included other arithmetic functions such as multiplication. Leibnitz, who with Newton also invented calculus, went on to propose the idea of computing with symbols as one of our most basic and general intellectual activities. He argued for a universal language in which one could solve any problem by calculating.

Early in the nineteenth century, the French engineer Joseph Jacquard (1752–1834) designed and constructed a machine that automated the process of weaving (see Figure 1.5c). Until then, each row in a weaving pattern had to be set up by hand, a quite tedious, error-prone process. Jacquard's loom was designed to accept input in the form of a set of punched cards. Each card described a row in a pattern of cloth. Although it was still an entirely mechanical device, Jacquard's loom possessed something that previous devices had lacked—the ability to execute an algorithm automatically. The set of cards expressed the algorithm or set of instructions that controlled the behavior of the loom. If the loom operator wanted to produce a different pattern, he just had to run the machine with a different set of cards.

The British mathematician Charles Babbage (1792–1871) took the concept of a programmable computer a step further by designing a model of a machine that, conceptually, bore a striking resemblance to a modern general-purpose computer. Babbage conceived his machine, which he called the Analytical Engine, as a mechanical device. His design called for four functional parts: a mill to perform arithmetic operations, a store to hold data and a program, an operator to run the instructions from punched cards, and an output to produce the results on punched cards. Sadly, Babbage's computer was never built. The project perished for lack of funds near the time when Babbage himself passed away.

In the last two decades of the nineteenth century, a U.S. Census Bureau statistician named Herman Hollerith (1860–1929) developed a machine that automated data processing for the U.S. Census. Hollerith's machine, which had the same component parts as Babbage's Analytical Engine, simply accepted a set of punched cards as input and then tallied and sorted the cards. His machine greatly shortened the time it took to produce statistical results on the U.S. population. Government and business organizations seeking to automate their data processing quickly adopted Hollerith's punched card machines. Hollerith was also one of the founders of a company that eventually became IBM (International Business Machines).

Also in the nineteenth century, the British secondary school teacher George Boole (1815–1864) developed a system of logic. This system consisted of a pair of

values, TRUE and FALSE, and a set of three primitive operations on these values, AND, OR, and NOT. Boolean logic eventually became the basis for designing the electronic circuitry to process binary information.

A half a century later, in the 1930s, the British mathematician Alan Turing (1912–1954) explored the theoretical foundations and limits of algorithms and computation. Turing’s essential contributions were to develop the concept of a universal machine that could be specialized to solve any computable problems, and to demonstrate that some problems are unsolvable by computers.

1.3.2 The First Electronic Digital Computers (1940–1950)

In the late 1930s, Claude Shannon (1916–2001), a mathematician and electrical engineer at MIT, wrote a classic paper titled “A Symbolic Analysis of Relay and Switching Circuits.” In this paper, he showed how operations and information in other systems, such as arithmetic, could be reduced to Boolean logic and then to hardware. For example, if the Boolean values TRUE and FALSE were written as the binary digits 1 and 0, one could write a sequence of logical operations that computes the sum of two strings of binary digits. All that was required to build an electronic digital computer was the ability to represent binary digits as on/off switches and to represent the logical operations in other circuitry.

The needs of the combatants in World War II pushed the development of computer hardware into high gear. Several teams of scientists and engineers in the United States, England, and Germany independently created the first generation of general-purpose digital electronic computers during the 1940s. All of these scientists and engineers used Shannon’s innovation of expressing binary digits and logical operations in terms of electronic switching devices. Among these groups was a team at Harvard University under the direction of Howard Aiken. Their computer, called the Mark I, became operational in 1944 and did mathematical work for the U.S. Navy during the war. The Mark I was considered an electromechanical device, because it used a combination of magnets, relays, and gears to store and process data.

Another team under J. Presper Eckert and John Mauchly, at the University of Pennsylvania, produced a computer called the ENIAC (Electronic Numerical Integrator and Calculator). The ENIAC calculated ballistics tables for the artillery of the U.S. Army toward the end of the war. Because the ENIAC used entirely electronic components, it was almost a thousand times faster than the Mark I.

Two other electronic digital computers were completed a bit earlier than the ENIAC. They were the ABC (Atanasoff-Berry Computer), built by John Atanasoff and Clifford Berry at Iowa State University in 1942, and the Colossus,

constructed by a group working under Alan Turing in England in 1943. The ABC was created to solve systems of simultaneous linear equations. Although the ABC's function was much narrower than that of the ENIAC, the ABC is now regarded as the first electronic digital computer. The Colossus, whose existence had been top secret until recently, was used to crack the powerful German Enigma code during the war.

The first electronic digital computers, sometimes called **mainframe computers**, consisted of vacuum tubes, wires, and plugs, and filled entire rooms. Although they were much faster than people at computing, by our own current standards, they were extraordinarily slow and prone to breakdown. Moreover, the early computers were extremely difficult to program. To enter or modify a program, a team of workers had to rearrange the connections among the vacuum tubes by unplugging and replugging the wires. Each program was loaded by literally hardwiring it into the computer. With thousands of wires involved, it was easy to make a mistake.

The memory of these first computers stored only data, not the program that processed the data. As we have seen, the idea of a stored program first appeared 100 years earlier in Jacquard's loom and in Babbage's design for the Analytical Engine. In 1946, John von Neumann realized that the instructions of the programs could also be stored in binary form in an electronic digital computer's memory. His research group at Princeton developed one of the first modern stored-program computers.

Although the size, speed, and applications of computers have changed dramatically since those early days, the basic architecture and design of the electronic digital computer have remained remarkably stable.

1.3.3 The First Programming Languages (1950–1965)

The typical computer user now runs many programs, made up of millions of lines of code, that perform what would have seemed like magical tasks 20 or 30 years ago. But the first digital electronic computers had no software as we think of it today. The machine code for a few relatively simple and small applications had to be loaded by hand. As the demand for larger and more complex applications grew, so did the need for tools to expedite the programming process.

In the early 1950s, computer scientists realized that a symbolic notation could be used instead of machine code, and the first **assembly languages** appeared. The programmers would enter mnemonic codes for operations, such as ADD and OUTPUT, and for data variables, such as SALARY and RATE, at a **keypunch machine**. The keystrokes punched a set of holes in a small card for each instruction. The programmers then carried their stacks of cards to a system

operator, who placed them in a device called a **card reader**. This device translated the holes in the cards to patterns in the computer's memory. A program called an **assembler** then translated the application programs in memory to machine code, and they were executed.

Programming in assembly language was a definite improvement over programming in machine code. The symbolic notation used in assembly languages was easier for people to read and understand. Another advantage was that the assembler could catch some programming errors before the program actually executed. However, the symbolic notation still appeared a bit arcane when compared with the notations of conventional mathematics. To remedy this problem, John Backus, a programmer working for IBM, developed FORTRAN (Formula Translation Language) in 1954. Programmers, many of whom were mathematicians, scientists, and engineers, could now use conventional algebraic notation. FORTRAN programmers still entered their programs on a keypunch machine, but the computer executed them after they were translated to machine code by a **compiler**.

FORTRAN was considered ideal for numerical and scientific applications. However, expressing the kind of data used in data processing—in particular, textual information—was difficult. For example, FORTRAN was not practical for processing information that included people's names, addresses, Social Security numbers, and the financial data of corporations and other institutions. In the early 1960s, a team led by Rear Admiral Grace Murray Hopper developed COBOL (Common Business Oriented Language) for data processing in the United States Government. Banks, insurance companies, and other institutions were quick to adopt its use in data-processing applications.

Also in the late 1950s and early 1960s, John McCarthy, a computer scientist at MIT, developed a powerful and elegant notation called LISP (List Processing) for expressing computations. Based on a theory of recursive functions (a subject covered in Chapter 6 of this book), LISP captured the essence of symbolic information processing. A student of McCarthy's, Stephen "Slug" Russell, coded the first **interpreter** for LISP in 1960. The interpreter accepted LISP expressions directly as inputs, evaluated them, and printed their results. In its early days, LISP was used primarily for laboratory experiments in an area of research known as **artificial intelligence**. More recently, LISP has been touted as an ideal language for solving any difficult or complex problems.

Although they were among the first high-level programming languages, FORTRAN and LISP have survived for decades. They have undergone many modifications to improve their capabilities and have served as models for the development of many other programming languages. COBOL, by contrast, is no longer in active use but has survived mainly in the form of legacy programs that must still be maintained.

These new, high-level programming languages had one feature in common: **abstraction**. In science or any other area of enquiry, an abstraction allows human beings to reduce complex ideas or entities to simpler ones. For example, a set of 10 assembly language instructions might be replaced with an equivalent algebraic expression that consists of only five symbols in FORTRAN. Put another way, any time you can say more with less, you are using an abstraction. The use of abstraction is also found in other areas of computing, such as hardware design and information architecture. The complexities don't actually go away, but the abstractions hide them from view. The suppression of distracting complexity with abstractions allows computer scientists to conceptualize, design, and build ever more sophisticated and complex systems.

1.3.4

Integrated Circuits, Interaction, and Timesharing (1965–1975)

In the late 1950s, the vacuum tube gave way to the **transistor** as the mechanism for implementing the electronic switches in computer hardware. As a **solid-state device**, the transistor was much smaller, more reliable, more durable, and less expensive to manufacture than a vacuum tube. Consequently, the hardware components of computers generally became smaller in physical size, more reliable, and less expensive. The smaller and more numerous the switches became, the faster the processing and the greater the capacity of memory to store information.

The development of the **integrated circuit** in the early 1960s allowed computer engineers to build ever smaller, faster, and less expensive computer hardware components. They perfected a process of photographically etching transistors and other solid-state components onto very thin wafers of silicon, leaving an entire processor and memory on a single chip. In 1965, Gordon Moore, one of the founders of the computer chip manufacturer Intel, made a prediction that came to be known as **Moore's Law**. This prediction states that the processing speed and storage capacity of hardware will increase and its cost will decrease by approximately a factor of 2 every 18 months. This trend has held true for the past 40 years. For example, there were about 50 electrical components on a chip in 1965, whereas by 2000, a chip could hold over 40 million components. Without the integrated circuit, men would not have gone to the moon in 1969, and we would not have the powerful and inexpensive handheld devices that we now use on a daily basis.

Minicomputers the size of a large office desk appeared in the 1960s. The means of developing and running programs also were changing. Until then, a computer was typically located in a restricted area with a single human operator.

Programmers composed their programs on keypunch machines in another room or building. They then delivered their stacks of cards to the computer operator, who loaded them into a card reader, and compiled and ran the programs in sequence on the computer. Programmers then returned to pick up the output results, in the form of new stacks of cards or printouts. This mode of operation, also called **batch processing**, might cause a programmer to wait days for results, including error messages.

The increases in processing speed and memory capacity enabled computer scientists to develop the first **time-sharing operating system**. John McCarthy, the creator of the programming language LISP, recognized that a program could automate many of the functions performed by the human system operator. When memory, including magnetic secondary storage, became large enough to hold several users' programs at the same time, they could be scheduled for **concurrent processing**. Each process associated with a program would run for a slice of time and then yield the CPU to another process. All of the active processes would repeatedly cycle for a turn with the CPU until they finished.

Several users could now run their own programs simultaneously by entering commands at separate terminals connected to a single computer. As processor speeds continued to increase, each user gained the illusion that a time-sharing computer system belonged entirely to him or her.

By the late 1960s, programmers could enter program input at a terminal and also see program output immediately displayed on a **CRT (Cathode Ray Tube) screen**. Compared to its predecessors, this new computer system was both highly interactive and much more accessible to its users.

Many relatively small and medium-sized institutions, such as universities, were now able to afford computers. These machines were used not only for data processing and engineering applications, but also for teaching and research in the new and rapidly growing field of computer science.

1.3.5 Personal Computing and Networks (1975–1990)

In the mid-1960s, Douglas Engelbart, a computer scientist working at the Stanford Research Institute (SRI), first saw one of the ultimate implications of Moore's Law: eventually, perhaps within a generation, hardware components would become small enough and affordable enough to mass produce an individual computer for every human being. What form would these personal computers take, and how would their owners use them? Two decades earlier, in 1945, Engelbart had read an article in *The Atlantic Monthly* titled "As We May Think" that had already posed this question and offered some answers. The author, Vannevar Bush, a scientist at MIT, predicted that computing devices would serve

as repositories of information and, ultimately, of all human knowledge. Owners of computing devices would consult this information by browsing through it with pointing devices, and contribute information to the knowledge base almost at will. Engelbart agreed that the primary purpose of the personal computer would be to augment the human intellect, and he spent the rest of his career designing computer systems that would accomplish this goal.

During the late 1960s, Engelbart built the first pointing device or mouse. He also designed software to represent windows, icons, and pull-down menus on a **bit-mapped display screen**. He demonstrated that a computer user could not only enter text at the keyboard but could also directly manipulate the icons that represent files, folders, and computer applications on the screen.

But for Engelbart, personal computing did not mean computing in isolation. He participated in the first experiment to connect computers in a network, and he believed that soon people would use computers to communicate, share information, and collaborate on team projects.

Engelbart developed his first experimental system, which he called NLS (oNLine System) Augment, on a minicomputer at SRI. In the early 1970s, he moved to Xerox PARC (Palo Alto Research Center) and worked with a team under Alan Kay to develop the first desktop computer system. Called the Alto, this system had many of the features of Engelbart's Augment, as well as e-mail and a functioning hypertext (a forerunner of the World Wide Web). Kay's group also developed a programming language called Smalltalk, which was designed to create programs for the new computer and to teach programming to children. Kay's goal was to develop a personal computer the size of a large notebook, which he called the Dynabook. Unfortunately for Xerox, the company's management had more interest in photocopy machines than in the work of Kay's visionary research group. However, a young entrepreneur named Steve Jobs visited the Xerox lab and saw the Alto in action. In 1984, Apple Computer, the now-famous company founded by Steve Jobs, brought forth the Macintosh, the first successful mass-produced personal computer with a graphical user interface.

While Kay's group was busy building the computer system of the future in their research lab, dozens of hobbyists gathered near San Francisco to found the Homebrew Computer Club, the first personal computer users group. They met to share ideas, programs, hardware, and applications for personal computing. The first mass-produced personal computer, the Altair, appeared in 1975. The Altair contained Intel's 8080 processor, the first **microcomputer** chip. But from the outside, the Altair looked and behaved more like a miniature version of the early computers than the Alto. Programs and their input had to be entered by flipping switches, and output was displayed by a set of lights. However, the Altair was small enough for personal computing enthusiasts to carry home, and I/O devices eventually were invented to support the processing of text and sound.

The Osborne and the Kaypro were among the first mass-produced interactive personal computers. They boasted tiny display screens and keyboards, with floppy disk drives for loading system software, applications software, and users' data files. Early personal computing applications were word processors, spreadsheets, and games such as PacMan and SpaceWar!. These computers also ran CP/M (Control Program for Microcomputers), the first PC-based operating system.

In the early 1980s, a college dropout named Bill Gates and his partner Paul Allen built their own operating system software, which they called MS-DOS (Microsoft Disk Operating System). They then arranged a deal with the giant computer manufacturer IBM to supply MS-DOS for the new line of PCs that the company intended to mass-produce. This deal proved to be a very advantageous one for Gates' company, Microsoft. Not only did Microsoft receive a fee for each computer sold, but it also was able to get a head start on supplying applications software that would run on its operating system. Brisk sales of the IBM PC and its "clones" to individuals and institutions quickly made MS-DOS the world's most widely used operating system. Within a few years, Gates and Allen had become billionaires, and within a decade, Gates had become the world's richest man, a position he held for 13 straight years.

Also in the 1970s, the U.S. Government began to support the development of a network that would connect computers at military installations and research universities. The first such network, called ARPANET (Advanced Research Projects Agency Network), connected four computers at SRI, UCLA (University of California at Los Angeles), UC Santa Barbara, and the University of Utah. Bob Metcalfe, a researcher associated with Kay's group at Xerox, developed a software protocol called Ethernet for operating a network of computers. Ethernet allowed computers to communicate in a local area network (LAN) within an organization and also with computers in other organizations via a wide area network (WAN). By the mid 1980s, the ARPANET had grown into what we now call the Internet, connecting computers owned by large institutions, small organizations, and individuals all over the world.

1.3.6 Consultation, Communication, and Ubiquitous Computing (1990–Present)

In the 1990s, computer hardware costs continued to plummet, and processing speed and memory capacity skyrocketed. **Optical storage media**, such as compact discs (CDs) and digital video discs (DVDs), were developed for mass storage. The computational processing of images, sound, and video became feasible and widespread. By the end of the decade, entire movies were being shot or constructed

and played back using digital devices. The capacity to create lifelike three-dimensional animations of whole environments led to a new technology called **virtual reality**. New devices appeared, such as flatbed scanners and digital cameras, which could be used along with the more traditional microphone and speakers to support the input and output of almost any type of information.

Desktop and laptop computers now not only perform useful work but also give their users new means of personal expression. The past decade has seen the rise of computers as communication tools, with e-mail, instant messaging, bulletin boards, chat rooms, and the amazing World Wide Web. With the rise of wireless technology, all of these capabilities are now available almost everywhere on tiny, handheld devices. Computing is becoming ubiquitous, yet also less visible.

Perhaps the most interesting story from this period concerns Tim Berners-Lee, the creator of the World Wide Web. In the late 1980s, Berners-Lee, a theoretical physicist doing research at the CERN Institute in Geneva, Switzerland, began to develop some ideas for using computers to share information. Computer engineers had been linking computers to networks for several years, and it was already common in research communities to exchange files and send and receive e-mail around the world. However, the vast differences in hardware, operating systems, file formats, and applications still made it difficult for users who were not adept at programming to access and share this information. Berners-Lee was interested in creating a common medium for sharing information that would be easy to use, not only for scientists but also for any other person capable of manipulating a keyboard and mouse and viewing the information on a monitor.

Berners-Lee was familiar with Vannevar Bush's vision of a web-like consultation system, Engelbart's work on NLS Augment, and also with the first widely available hypertext systems. One of these systems, Apple Computer's Hypercard, broadened the scope of hypertext to **hypermedia**. Hypercard allowed authors to organize not just text but also images, sound, video, and executable applications into webs of linked information. However, a Hypercard database sat only on stand-alone computers; the links could not carry Hypercard data from one computer to another. Furthermore, the supporting software ran only on Apple's computers.

Berners-Lee realized that networks could extend the reach of a hypermedia system to any computers connected to the net, making their information available worldwide. To preserve its independence from particular operating systems, the new medium would need to have universal standards for distributing and presenting the information. To ensure this neutrality and independence, no private corporation or individual government could own the medium and dictate the standards.

Berners-Lee built the software for this new medium, which we now call the World Wide Web, in 1992. The software used many of the existing mechanisms for transmitting information over the Internet. People contribute information to

the Web by publishing files on computers known as **Web servers**. The Web server software on these computers is responsible for answering requests for viewing the information stored on the Web server. To view information on the Web, people use software called a **Web browser**. In response to a user's commands, a Web browser sends a request for information across the Internet to the appropriate Web server. The server responds by sending the information back to the browser's computer, called a **Web client**, where it is displayed or rendered in the browser.

Although Berners-Lee wrote the first Web server and Web browser software, he made two other, even more important contributions. First, he designed a set of rules, called HTTP (Hypertext Transfer Protocol), which allows any server and browser to talk to each other. Second, he designed a language, HTML (Hypertext Markup Language), which allows browsers to structure the information to be displayed on Web pages. He then made all of these resources available to anyone for free.

Berners-Lee's invention and gift of this universal information medium is a truly remarkable achievement. Today there are millions of Web servers in operation around the world. Anyone with the appropriate training and resources—companies, government, nonprofit organizations, and private individuals—can start up a new Web server or obtain space on one. Web browser software now runs not only on desktop and laptop computers, but also on handheld devices such as cell phones.

This concludes our not-so-brief overview of the history of computing. If you want to learn more about this history, consult the sources listed at the end of this chapter. We now turn to an introduction to programming in Python.

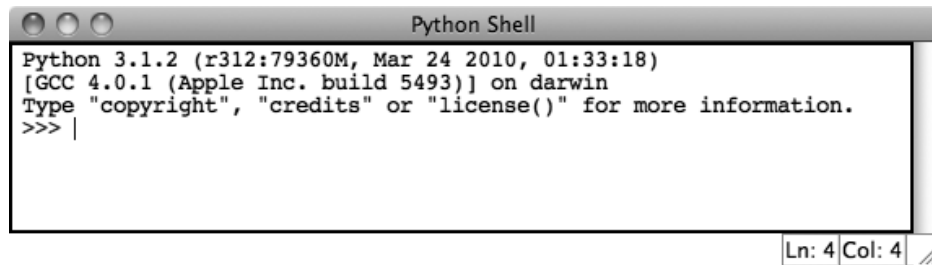
1.4 Getting Started with Python Programming

Guido van Rossum invented the Python programming language in the early 1990s. Python is a high-level, general-purpose programming language for solving problems on modern computer systems. The language and many supporting tools are free, and Python programs can run on any operating system. You can download Python, its documentation, and related materials from www.python.org. You can find instructions for downloading and installing Python in Appendix A. In this section, we show you how to create and run simple Python programs.

1.4.1 Running Code in the Interactive Shell

Python is an interpreted language, and you can run simple Python expressions and statements in an interactive programming environment called the **shell**. The easiest way to open a Python shell is to launch the IDLE. This is an integrated program

development environment that comes with the Python installation. When you do this, a window named **Python Shell** opens. Figure 1.6 shows a shell window on Mac OS X. A shell window running on a Windows system or a Linux system should look similar, if not identical, to this one.



[FIGURE 1.6] Python shell window

A shell window contains an opening message followed by the special symbol `>>>`, called a shell prompt. The cursor at the shell prompt waits for you to enter a Python command. Note that you can get immediate help by entering **help** at the shell prompt or selecting **Help** from the window's drop-down menu.

When you enter an expression or statement, Python evaluates it and displays its result, if there is one, followed by a new prompt. The next few lines show the evaluation of several expressions and statements. In this example, the results are displayed in italics, although they would not actually appear in italics on the computer screen.

```
>>> 3 + 4
7
>>> 3
3
>>> "Python is really cool!"
'Python is really cool!'
>>> name = "Ken Lambert"
>>> name
'Ken Lambert'
>>> "Hi there " + name
'Hi there Ken Lambert'
>>> print('Hi there')
Hi there
>>> print("Hi there", name)
Hi there Ken Lambert
>>>
```

To quit the Python shell, you can either select the window's close box or press the Control+D key combination.

The Python shell is useful for experimenting with short expressions or statements to learn new features of the language, as well as for consulting documentation on the language. The means of developing more complex and interesting programs are examined in the rest of this section.

1.4.2 Input, Processing, and Output

Most useful programs accept inputs from some source, process these inputs, and then finally output results to some destination. In terminal-based interactive programs, the input source is the keyboard, and the output destination is the terminal display. The Python shell itself is such a program; its inputs are Python expressions or statements. Its processing evaluates these items. Its outputs are the results displayed in the shell.

The programmer can also force the output of a value by using the **print** function. The simplest form for using this function looks like the following:

```
print(<expression>)
```

This example shows you the basic **syntax** (or grammatical rule) for using the **print** function. The angle brackets (the < and > symbols) enclose a type of phrase. In actual Python code, you would replace this syntactic form, including the angle brackets, with an example of that type of phrase. In this case, **<expression>** is shorthand for any Python expression.

When running the **print** function, Python first evaluates the expression and then displays its value. In the example shown earlier, **print** was used to display some text. The following is another example:

```
>>> print('Hi there')
Hi there
```

In this example, the text **'Hi there'** is the text that we want Python to display. In programming terminology, this piece of text is referred to as a string. In Python code, a string is always enclosed in quotation marks. However, the **print** function displays a string without the quotation marks.

You can also write a **print** function that includes two or more expressions separated by commas. In such a case, the **print** function evaluates the expressions and

displays their results, separated by single spaces, on one line. The syntax for a **print** statement with two or more expressions looks like the following:

```
print(<expression>, ... , <expression>)
```

Note the ellipsis in this syntax example. The ellipsis indicates that you could include multiple expressions after the first one. Whether it outputs one or multiple expressions, the **print** function always ends its output with a **newline**. In other words, it displays the values of the expressions, and then it moves the cursor to the next line on the console window.

To begin the next output on the same line as the previous one, you can place the expression **end=""**, which says end the line with an empty string, at the end of the list of expressions, as follows:

```
print(<expression>, end="")
```

As you create programs in Python, you'll often want your programs to ask the user for input. You can do this by using the **input** function. This function causes the program to stop and wait for the user to enter a value from the keyboard. When the user presses the return or enter key, the function accepts the input value and makes it available to the program. A program that receives an input value in this manner typically saves it for further processing.

The following example receives an input string from the user and saves it for further processing. The user's input is in italics.

```
>>> name = input("Enter your name: ")
Enter your name: Ken Lambert
>>> name
'Ken Lambert'
>>> print(name)
Ken Lambert
>>>
```

The **input** function does the following:

- 1 Displays a prompt for the input. In this example, the prompt is **"Enter your name: "**.
- 2 Receives a string of keystrokes, called characters, entered at the keyboard and returns the string to the shell.

How does the **input** function know what to use as the prompt? The text in parentheses, "**Enter your name:** ", is an argument for the **input** function that tells it what to use for the prompt. An **argument** is a piece of information that a function needs to do its work.

The string returned by the function in our example is saved by assigning it to the variable **name**. The form of an assignment statement with the **input** function is the following:

```
<variable identifier> = input(<a string prompt>)
```

A **variable identifier**, or **variable** for short, is just a name for a value. When a variable receives its value in an input statement, the variable then refers to this value. If the user enters the name "**Ken Lambert**" in our last example, the value of the variable **name** can be viewed as follows:

```
>>> name
'Ken Lambert'
```

The **input** function always builds a string from the user's keystrokes and returns it to the program. After inputting strings that represent numbers, the programmer must convert them from strings to the appropriate numeric types. In Python, there are two **type conversion functions** for this purpose, called **int** (for integers) and **float** (for floating-point numbers). The next session inputs two integers and displays their sum:

```
>>> first = int(input("Enter the first number: "))
Enter the first number: 23
>>> second = int(input("Enter the second number: "))
Enter the second number: 44
>>> print("The sum is", first + second)
The sum is 67
>>>
```

Note that the **int** function is called with each result returned by the **input** function. The two numbers are added, and then their sum is output. Table 1.1 summarizes the functions introduced in this subsection.

FUNCTION	WHAT IT DOES
<code>float(<a string of digits>)</code>	Converts a string of digits to a floating-point value.
<code>int(<a string of digits>)</code>	Converts a string of digits to an integer value.
<code>input(<a string prompt>)</code>	Displays the string prompt and waits for keyboard input. Returns the string of characters entered by the user.
<code>print(<expression>, ... , <expression>)</code>	Evaluates the expressions and displays them, separated by one space, in the console window.
<code><string 1> + <string 2></code>	Glues the two strings together and returns the result.

[TABLE 1.1] Basic Python functions for input and output

1.4.3 Editing, Saving, and Running a Script

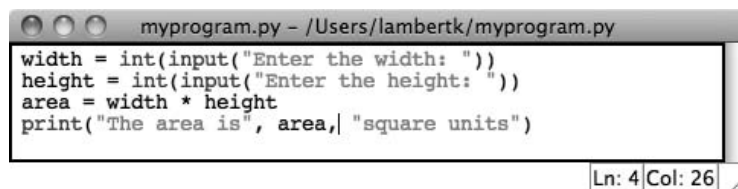
While it is easy to try out short Python expressions and statements interactively at a shell prompt, it is more convenient to compose, edit, and save longer, more complex programs in files. We can then run these program files or **scripts** either within IDLE or from the operating system's command prompt without opening IDLE. Script files are also the means by which Python programs are distributed to others. Most important, as you know from writing term papers, files allow you to save, safely and permanently, many hours of work.

To compose and execute programs in this manner, you perform the following steps:

- 1 Select the option **New Window** from the **File** menu of the shell window.
- 2 In the new window, enter Python expressions or statements on separate lines, in the order in which you want Python to execute them.
- 3 At any point, you may save the file by selecting **File/Save**. If you do this, you should use a **.py** extension. For example, your first program file might be named **myprogram.py**.
- 4 To run this file of code as a Python script, select **Run Module** from the **Run** menu or press the F5 key (Windows) or the Control+F5 key (Mac or Linux).

The command in Step 4 reads the code from the saved file and executes it. If Python executes any **print** functions in the code, you will see the outputs as usual in the shell window. If the code requests any inputs, the interpreter will pause to allow you to enter them. Otherwise, program execution continues invisibly behind the scenes. When the interpreter has finished executing the last instruction, it quits and returns you to the shell prompt.

Figure 1.7 shows an IDLE window containing a complete script that prompts the user for the width and height of a rectangle, computes its area, and outputs the result:

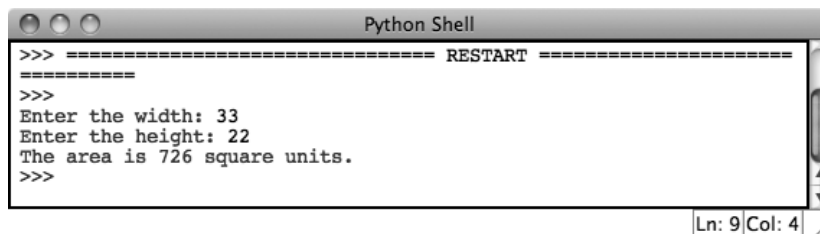


```
width = int(input("Enter the width: "))
height = int(input("Enter the height: "))
area = width * height
print("The area is", area, "square units")
```

Ln: 4 Col: 26

[FIGURE 1.7] Python script in an IDLE window

When the script is run from the IDLE window, it produces the interaction with the user in the shell window shown in Figure 1.8.



```
>>> ===== RESTART =====
>>>
Enter the width: 33
Enter the height: 22
The area is 726 square units.
>>>
```

Ln: 9 Col: 4

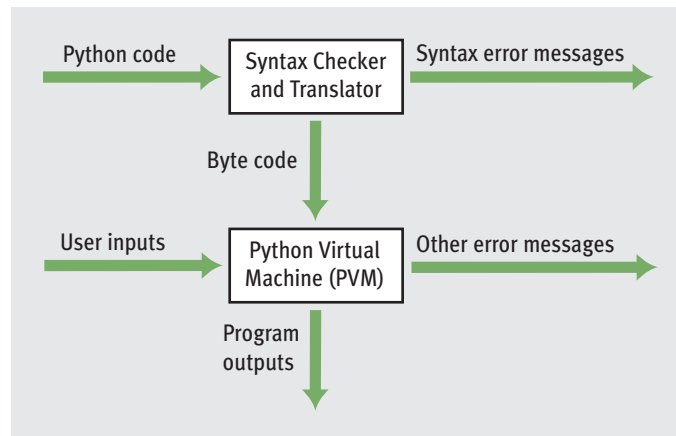
[FIGURE 1.8] Interaction with a script in a shell window

This can be a slightly less interactive way of executing programs than entering them directly at Python's interpreter prompt. However, running the script from the IDLE window will allow you to construct some complex programs, test them, and save them in **program libraries** that you can reuse or share with others.

1.4.4

Behind the Scenes: How Python Works

Whether you are running Python code as a script or interactively in a shell, the Python interpreter does a great deal of work to carry out the instructions in your program. This work can be broken into a series of steps, as shown in Figure 1.9.



[FIGURE 1.9] Steps in interpreting a Python program

- 1 The interpreter reads a Python expression or statement, also called the **source code**, and verifies that it is well formed. In this step, the interpreter behaves like a strict English teacher who rejects any sentence that does not adhere to the grammar rules, or syntax, of the language. As soon as the interpreter encounters such an error, it halts translation with an error message.
- 2 If a Python expression is well formed, the interpreter then translates it to an equivalent form in a low-level language called **byte code**. When the interpreter runs a script, it completely translates it to byte code.
- 3 This byte code is next sent to another software component, called the **Python virtual machine (PVM)**, where it is executed. If another error occurs during this step, execution also halts with an error message.

1.4 Exercises

- 1 Describe what happens when the programmer enters the string **"Greetings!"** in the Python shell.
- 2 Write a line of code that prompts the user for his or her name and saves the user's input in a variable called **name**.
- 3 What is a Python script?
- 4 Explain what goes on behind the scenes when your computer runs a Python program.

1.5 Detecting and Correcting Syntax Errors

Programmers inevitably make typographical errors when editing programs, and the Python interpreter will nearly always detect them. Such errors are called syntax errors. The term syntax refers to the rules for forming sentences in a language. When Python encounters a syntax error in a program, it halts execution with an error message. The following sessions with the Python shell show several types of syntax errors and the corresponding error messages:

```
>>> length = int(input("Enter the length: "))
Enter the length: 44

>>> print(lenth)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
NameError: name 'lenth' is not defined
```

The first statement assigns an input value to the variable **length**. The next statement attempts to print the value of the variable **lenth**. Python responds that this name is not defined. Although the programmer might have *meant* to write the variable **length**, Python can read only what the programmer *actually entered*. This is a good example of the rule that a computer can read *only* the instructions it receives, not the instructions we intend to give it.

The next statement attempts to print the value of the correctly spelled variable, but Python still generates an error message.

```
>>> print length
      File "<pyshell#1>", line 1
        print length
          ^
SyntaxError: unexpected indent
```

In this error message, Python explains that this line of code is unexpectedly indented. In fact, there is an extra space before the word **print**. Indentation is significant in Python code. Each line of code entered at a shell prompt or in a script must begin in the leftmost column, with no leading spaces. The only exception to this rule occurs in control statements and definitions, where nested statements must be indented one or more spaces.

You might think that it would be painful to keep track of indentation in a program. However, in compensation, the Python language is much simpler than other programming languages. Consequently, there are fewer types of syntax errors to encounter and correct, and a lot less syntax for you to learn!

In our final example, the programmer attempts to add two numbers, but forgets to include the second one:

```
>>> 3 +  
3 +  
SyntaxError: invalid syntax
```

In later chapters, you will learn more about other kinds of program errors and how to repair the code that generates them.

1.5 Exercises

- 1 Suppose your script attempts to print the value of a variable that has not yet been assigned a value. How does the Python interpreter react?
- 2 Miranda has forgotten to complete an arithmetic expression before the end of a line of code. How will the Python interpreter react?
- 3 Why does Python code generate fewer types of syntax errors than code in other programming languages?

Suggestions for Further Reading

John Battelle, *The Search: How Google and Its Rivals Rewrote the Rules of Business and Transformed Our Culture* (New York: Portfolio Trade, 2006).

Tim Berners-Lee, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web* (New York: Harper-Collins, 2000).

Paul Graham, *Hackers and Painters: Big Ideas from the Computer Age* (Sebastopol, CA: O'Reilly, 2004).

Katie Hafner and Matthew Lyon, *Where Wizards Stay Up Late: The Origins of the Internet* (New York: Simon and Schuster, 1996).

Michael E. Hobart and Zachary S. Schiffman, *Information Ages: Literacy, Numeracy, and the Computer Revolution* (Baltimore: The Johns Hopkins University Press, 1998).

Georges Ifrah, *The Universal History of Computing: From the Abacus to the Quantum Computer* (New York: John Wiley & Sons, Inc., 2001).

John Markoff, *What the Doormouse Said: How the Sixties Counterculture Shaped the Personal Computer Industry* (New York: Viking, 2005).

Summary

- One of the most fundamental ideas of computer science is the algorithm. An algorithm is a sequence of instructions for solving a problem. A computing agent can carry out these instructions to solve a problem in a finite amount of time.
- Another fundamental idea of computer science is information processing. Practically any relationship among real-world objects can be represented as information or data. Computing agents manipulate information and transform it by following the steps described in algorithms.
- Real computing agents can be constructed out of hardware devices. These consist of a central processing unit (CPU), a memory, and input and output devices. The CPU contains circuitry that executes the instructions described by algorithms. The memory contains switches that represent binary digits. All information stored in memory is represented in binary form. Input devices such as a keyboard and flatbed scanner and output devices such as a monitor and speakers transmit information between the computer's memory and the external world. These devices also transfer information between a binary form and a form that human beings can use.
- Some real computers, such as those in wristwatches and cell phones, are specialized for a small set of tasks, whereas a desktop or laptop computer is a general-purpose problem-solving machine.
- Software provides the means whereby different algorithms can be run on a general-purpose hardware device. The term “software” can refer to editors and interpreters for developing programs, an operating system for managing hardware devices, user interfaces for communicating with human users, and applications such as word processors, spreadsheets, database managers, games, and media-processing programs.

- Software is written in programming languages. Languages such as Python are high level; they resemble English and allow authors to express their algorithms clearly to other people. A program called an interpreter translates a Python program to a lower-level form that can be executed on a real computer.
- The Python shell provides a command prompt for evaluating and viewing the results of Python expressions and statements. IDLE is an integrated development environment that allows the programmer to save programs in files and load them into a shell for testing.
- Python scripts are programs that are saved in files and run from a terminal command prompt. An interactive script consists of a set of input statements, statements that process these inputs, and statements that output the results.
- When a Python program is executed, it is translated into byte code. This byte code is then sent to the Python virtual machine (PVM) for further interpretation and execution.
- Syntax is the set of rules for forming correct expressions and statements in a programming language. When the interpreter encounters a syntax error in a Python program, it halts execution with an error message. Two examples of syntax errors are a reference to a variable that does not yet have a value and an indentation that is unexpected.

REVIEW QUESTIONS

- 1 Which of the following are examples of algorithms?
 - a A dictionary
 - b A recipe
 - c A set of instructions for putting together a utility shed
 - d The spelling checker of a word processor
- 2 Which of the following contain information?
 - a My grandmother's china cabinet
 - b An audio CD
 - c A refrigerator
 - d A book
 - e A running computer
- 3 Which of the following are general-purpose computing devices?
 - a A cell phone
 - b A portable music player
 - c A laptop computer
 - d A programmable thermostat
- 4 Which of the following are input devices?
 - a Speakers
 - b Microphone
 - c Printers
 - d A mouse
- 5 Which of the following are output devices?
 - a A digital camera
 - b A keyboard
 - c A flatbed scanner
 - d A monitor

- 6 What is the purpose of the CPU?
 - a Store information
 - b Receive inputs from the human user
 - c Decode and execute instructions
 - d Send output to the human user
- 7 Which of the following translates and executes instructions in a programming language?
 - a A compiler
 - b A text editor
 - c A loader
 - d An interpreter
- 8 Which of the following outputs data in a Python program?
 - a The input statement
 - b The assignment statement
 - c The print statement
 - d The main function
- 9 What is IDLE used to do?
 - a Edit Python programs
 - b Save Python programs to files
 - c Run Python programs
 - d All of the above
- 10 What is the set of rules for forming sentences in a language called?
 - a Semantics
 - b Pragmatics
 - c Syntax
 - d Logic

PROJECTS

- 1 Open a Python shell, enter the following expressions, and observe the results:

```
a 8
b 8 * 2
c 8 ** 2
d 8 / 12
e 8 // 12
f 8 / 0
```

- 2 Write a Python program that prints (displays) your name, address, and telephone number.
- 3 Evaluate the following code at a shell prompt: **print("Your name is", name)**. Then assign **name** an appropriate value, and evaluate the statement again.
- 4 Open an IDLE window, and enter the program from Figure 1.7 that computes the area of a rectangle. Load the program into the shell by pressing the F5 key, and correct any errors that occur. Test the program with different inputs by running it at least three times.
- 5 Modify the program of Project 4 to compute the area of a triangle. Issue the appropriate prompts for the triangle's base and height, and change the names of the variables appropriately. Then, use the formula **.5 * base * height** to compute the area. Test the program from an IDLE window.
- 6 Write and test a program that computes the area of a circle. This program should request a number representing a radius as input from the user. It should use the formula **3.14 * radius ** 2** to compute the area, and output this result suitably labeled.
- 7 Write and test a program that accepts the user's name (as text) and age (as a number) as input. The program should output a sentence containing the user's name and age.

- 8 Enter an input statement using the **input** function at the shell prompt. When the prompt asks you for input, enter a number. Then, attempt to add 1 to that number, observe the results, and explain what happened.
- 9 Enter an input statement using the **input** function at the shell prompt. When the prompt asks you for input, enter your first name, observe the results, and explain what happened.
- 10 Enter the expression **help()** at the shell prompt. Follow the instructions to browse the topics and modules.

[CHAPTER]

2

SOFTWARE DEVELOPMENT, Data Types, and Expressions

After completing this chapter, you will be able to

- Describe the basic phases of software development: analysis, design, coding, and testing
- Use strings for the terminal input and output of text
- Use integers and floating-point numbers in arithmetic operations
- Construct arithmetic expressions
- Initialize and use variables with appropriate names
- Import functions from library modules
- Call functions with arguments and use returned values appropriately
- Construct a simple Python program that performs inputs, calculations, and outputs
- Use docstrings to document Python programs

This chapter begins with a discussion of the software development process, followed by a case study in which we walk through the steps of program analysis, design, coding, and testing. We also examine the basic elements from which programs are composed. These include the data types for text and numbers and the expressions that manipulate them. The chapter concludes with an introduction to the use of functions and modules in simple programs.

2.1

The Software Development Process

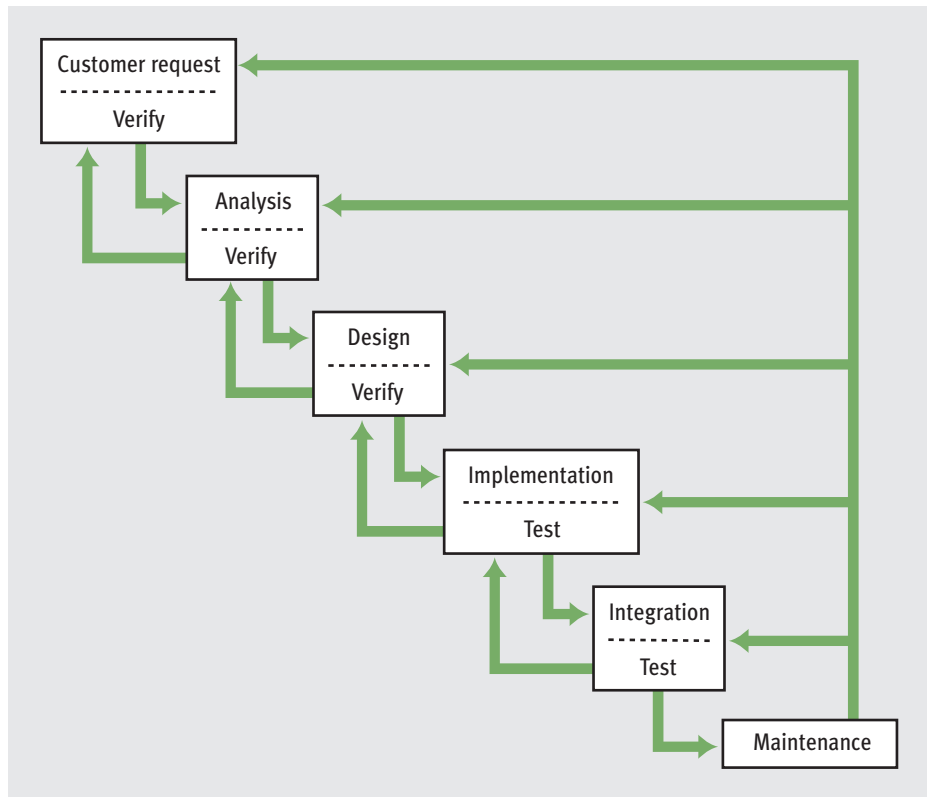
There is much more to programming than writing lines of code, just as there is more to building houses than pounding nails. The “more” consists of organization and planning, and various conventions for diagramming those plans. Computer scientists refer to the process of planning and organizing a program as **software development**. There are several approaches to software development. One version is known as the **waterfall model**.

The waterfall model consists of several phases:

- 1 **Customer request**—In this phase, the programmers receive a broad statement of a problem that is potentially amenable to a computerized solution. This step is also called the user requirements phase.
- 2 **Analysis**—The programmers determine what the program will do. This is sometimes viewed as a process of clarifying the specifications for the problem.
- 3 **Design**—The programmers determine how the program will do its task.
- 4 **Implementation**—The programmers write the program. This step is also called the coding phase.
- 5 **Integration**—Large programs have many parts. In the integration phase, these parts are brought together into a smoothly functioning whole, usually not an easy task.
- 6 **Maintenance**—Programs usually have a long life; a lifespan of 5 to 15 years is common for software. During this time, requirements change, errors are detected, and minor or major modifications are made.

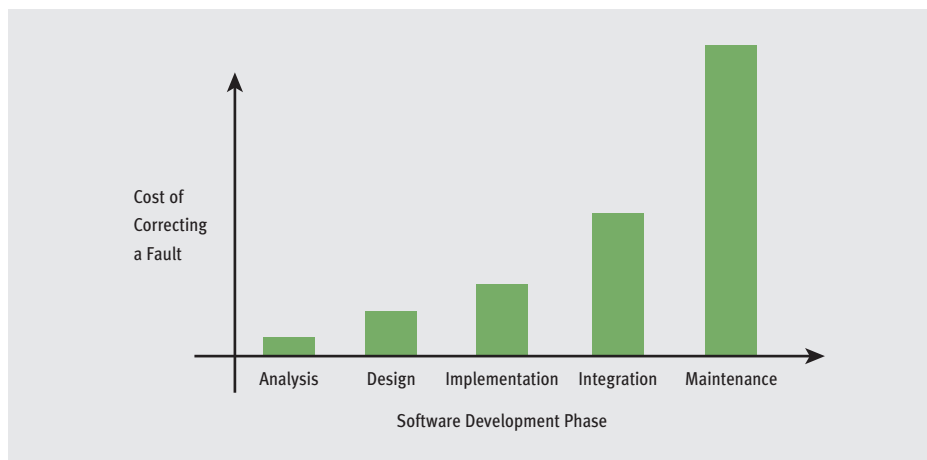
The phases of the waterfall model are shown in Figure 2.1. As you can see, the figure resembles a waterfall, in which the results of each phase flow down to the next. However, a mistake detected in one phase often requires the developer to back up and redo some of the work in the previous phase. Modifications made during maintenance also require backing up to earlier phases.

Although the diagram depicts distinct phases, this does not mean that developers must analyze and design a complete system before coding it. Modern software development is usually **incremental** and **iterative**. This means that analysis and design may produce a rough draft, skeletal version, or **prototype** of a system for coding, and then back up to earlier phases to fill in more details after some testing. For purposes of introducing this process, however, we treat these phases as distinct.



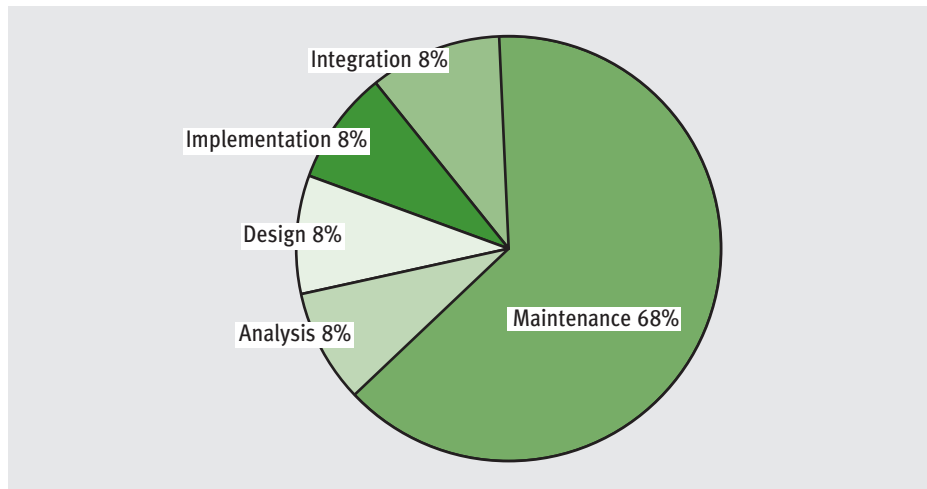
[FIGURE 2.1] The waterfall model of the software development process

Programs rarely work as hoped the first time they are run; hence, they should be subjected to extensive and careful testing. Many people think that testing is an activity that applies only to the implementation and integration phases; however, you should scrutinize the outputs of each phase carefully. Keep in mind that mistakes found early are much less expensive to correct than those found late. Figure 2.2 illustrates some relative costs of repairing mistakes when found in different phases. These are not just financial costs but also costs in time and effort.



[FIGURE 2.2] Relative costs of repairing mistakes that are found in different phases

Keep in mind that the cost of developing software is not spread equally over the phases. The percentages shown in Figure 2.3 are typical.



[FIGURE 2.3] Percentage of total cost incurred in each phase of the development process

You might think that implementation takes the most time and, therefore, costs the most. However, as you can see in Figure 2.3, maintenance is actually the most expensive part of software development. The cost of maintenance can be reduced by careful analysis, design, and implementation.

As you read this book and begin to sharpen your programming skills, you should remember two points:

- 1 There is more to software development than writing code.
- 2 If you want to reduce the overall cost of software development, write programs that are easy to maintain. This requires thorough analysis, careful design, and a good coding style. We will have more to say about coding styles throughout the book.

2.1 Exercises

- 1 List four phases of the software development process, and explain what they accomplish.
- 2 Jack says that he will not bother with analysis and design but proceed directly to coding his programs. Why is that not a good idea?

2.2 Case Study: Income Tax Calculator

Most of the chapters in this book include a case study that illustrates the software development process. This approach may seem overly elaborate for small programs, but it scales up well when programs become larger. The first case study develops a program that calculates income tax.

Each year, nearly everyone with an income faces the unpleasant task of computing his or her income tax return. If only it could be done as easily as suggested in this case study. We start with the customer request phase.

2.2.1 Request

The customer requests a program that computes a person's income tax.

2.2.2 Analysis

Analysis often requires the programmer to learn some things about the problem domain, in this case, the relevant tax law. For the sake of simplicity, let's assume the following tax laws:

- All taxpayers are charged a flat tax rate of 20%.
- All taxpayers are allowed a \$10,000 standard deduction.
- For each dependent, a taxpayer is allowed an additional \$3,000 deduction.
- Gross income must be entered to the nearest penny.
- The income tax is expressed as a decimal number.

Another part of analysis determines what information the user will have to provide. In this case, the user inputs are gross income and number of dependents. The program calculates the income tax based on the inputs and the tax law and then displays the income tax. Figure 2.4 shows the proposed terminal-based interface. Characters in *italics* indicate user inputs. The program prints the rest. The inclusion of an interface at this point is a good idea because it allows the customer and the programmer to discuss the intended program's behavior in a context understandable to both.

```
Enter the gross income: 150000.00  
Enter the number of dependents: 3  
The income tax is $26200.00
```

[FIGURE 2.4] The user interface for the income tax calculator

2.2.3 Design

During analysis, we specify what a program is going to do. In the next phase, design, we describe how the program is going to do it. This usually involves writing an algorithm. In Chapter 1, we showed how to write algorithms in ordinary English. In fact, algorithms are more often written in a somewhat stylized version of English called **pseudocode**. Here is the pseudocode for our income tax program:

```
Input the gross income and number of dependents  
Compute the taxable income using the formula  
Taxable income = gross income - 10000 - (3000 * number of dependents)  
Compute the income tax using the formula
```


Tax = taxable income * 0.20
Print the tax

Although there are no precise rules governing the syntax of pseudocode, in your pseudocode you should strive to describe the essential elements of the program in a clear and concise manner. Note that this pseudocode closely resembles Python code, so the transition to the coding step should be straightforward.

2.2.4 Implementation (Coding)

Given the preceding pseudocode, an experienced programmer would now find it easy to write the corresponding Python program. For a beginner, on the other hand, writing the code can be the most difficult part of the process. Although the program that follows is simple by most standards, do not expect to understand every bit of it at first. The rest of this chapter explains the elements that make it work and much more.

```
"""
Program: taxform.py
Author: Ken Lambert

Compute a person's income tax.

1. Significant constants
    tax rate
    standard deduction
    deduction per dependent
2. The inputs are
    gross income
    number of dependents
3. Computations:
    taxable income = gross income - the standard deduction -
                    a deduction for each dependent
    income tax = is a fixed percentage of the taxable income
4. The outputs are
    the income tax
"""

# Initialize the constants
TAX_RATE = 0.20
STANDARD_DEDUCTION = 10000.0
DEPENDENT_DEDUCTION = 3000.0
```

continued

```

# Request the inputs
grossIncome = float(input("Enter the gross income: "))
numDependents = int(input("Enter the number of dependents: "))

# Compute the income tax
taxableIncome = grossIncome - STANDARD_DEDUCTION - \
    DEPENDENT_DEDUCTION * numDependents
incomeTax = taxableIncome * TAX_RATE

# Display the income tax
print("The income tax is $" + str(incomeTax))

```

2.2.5 Testing

Our income tax program can run as a script from an IDLE window. If there are no syntax errors, we will be able to enter a set of inputs and view the results. However, a single run without syntax errors and with correct outputs provides just a slight indication of a program's correctness. Only thorough testing can build confidence that a program is working correctly. Testing is a deliberate process that requires some planning and discipline on the programmer's part. It would be much easier to turn the program in after the first successful run to meet a deadline or to move on to the next assignment. But your grade, your job, or people's lives might be affected by the slipshod testing of software.

Testing can be performed easily from an IDLE window. The programmer just loads the program repeatedly into the shell and enters different sets of inputs. The real challenge is coming up with sets of inputs that can reveal an error. An error at this point, also called a **logic error** or a **design error**, is an unexpected output.

A **correct program** produces the expected output for any legitimate input. The tax calculator's analysis does not provide a specification of what inputs are legitimate, but common sense indicates that they would be numbers greater than or equal to 0. Some of these inputs will produce outputs that are less than 0, but we will assume for now that these outputs are expected. Even though the range of the input numbers on a computer is finite, testing all of the possible combinations of inputs would be impractical. The challenge is to find a smaller set of inputs, called a **test suite**, from which we can conclude that the program will likely be correct for all inputs. In the tax program, we try inputs of 0, 1, and 2 for the number of dependents. If the program works correctly with these, we can assume that it will work correctly with larger values. The test inputs for the gross income are a number equal to the standard deduction and a number twice that amount (10000 and 20000, respectively). These two values will show the cases of a minimum

expected tax (0) and expected taxes that are less than or greater than 0. The program is run with each possible combination of the two inputs. Table 2.1 shows the possible combinations of inputs and the expected outputs in the test suite.

NUMBER OF DEPENDENTS	GROSS INCOME	EXPECTED TAX
0	10000	0
1	10000	−600
2	10000	−1200
0	20000	2000
1	20000	1400
2	20000	800

[TABLE 2.1] The test suite for the tax calculator program

If there is a logic error in the code, it will almost certainly be caught using these data. Note that the negative outputs are not considered errors. We will see how to prevent such computations in the next chapter.

2.3

Strings, Assignment, and Comments

Text processing is by far the most common application of computing. E-mail, text messaging, Web pages, and word processing all rely on and manipulate data consisting of strings of characters. This section introduces the use of strings for the output of text and the documentation of Python programs. We begin with an introduction to data types in general.

2.3.1

Data Types

In the real world, we use data all the time without bothering to consider what kind of data we’re using. For example, consider this sentence: “In 2007, Micaela paid \$120,000 for her house at 24 East Maple Street.” This sentence includes at least four pieces of data—a name, a date, a price, and an address—but of course you don’t have to stop to think about that before you utter the sentence. You certainly don’t have to stop to consider that the name consists only of text characters, the date and house price are numbers, and so on. However, when we use data in a computer program, we do need to keep in mind the type of data we’re

using. We also need to keep in mind what we can do with (what operations can be performed on) particular data.

In programming, a **data type** consists of a set of values and a set of operations that can be performed on those values. A **literal** is the way a value of a data type looks to a programmer. The programmer can use a literal in a program to mention a data value. When the Python interpreter evaluates a literal, the value it returns is simply that literal. Table 2.2 shows example literals of several Python data types.

TYPE OF DATA	PYTHON TYPE NAME	EXAMPLE LITERALS
Integers	<code>int</code>	<code>-1</code> , <code>0</code> , <code>1</code> , <code>2</code>
Real numbers	<code>float</code>	<code>-0.55</code> , <code>.3333</code> , <code>3.14</code> , <code>6.0</code>
Character strings	<code>str</code>	<code>"Hi"</code> , <code>""</code> , <code>'A'</code> , <code>'66'</code>

[TABLE 2.2] Literals for some Python data types

The first two data types listed in Table 2.2, **int** and **float**, are called **numeric data types**, because they represent numbers. You’ll learn more about numeric data types later in this chapter. For now, we will focus on character strings—which are often referred to simply as strings.

2.3.2 String Literals

In Python, a string literal is a sequence of characters enclosed in single or double quotation marks. The following session with the Python shell shows some example strings:

```
>>> 'Hello there!'
'Hello there!'
>>> "Hello there!"
'Hello there!'
>>> ''
''
>>> ""
''
>>>
```

The last two string literals ('' and "") represent the **empty string**. Although it contains no characters, the empty string is a string nonetheless. Note that the empty string is different from a string that contains a single blank space character, " ".

Double-quoted strings are handy for composing strings that contain single quotation marks or apostrophes. Here is a self-justifying example:

```
>>> "I'm using a single quote in this string!"
"I'm using a single quote in this string!"
>>> print("I'm using a single quote in this string!")
I'm using a single quote in this string!
>>>
```

Note that the **print** function displays the nested quotation mark but not the enclosing quotation marks. A double quotation mark can also be included in a string literal if one uses the single quotation marks to enclose the literal.

When you write a string literal in Python code that will be displayed on the screen as output, you need to determine whether you want to output the string as a single line or as a multi-line paragraph. If you want to output the string as a single line, you have to include the entire string literal (including its opening and closing quotation marks) in the same line of code. Otherwise, a syntax error will occur. To output a paragraph of text that contains several lines, you could use a separate **print** function call for each line. However, it is more convenient to enclose the entire string literal, line breaks and all, within three consecutive quotation marks (either single or double) for printing. The next session shows how this is done:

```
>>> print("""This very long sentence extends all the way to
the next line.""")
This very long sentence extends all the way to
the next line.
```

Note that the first line in the output ends exactly where the first line ends in the code.

When you evaluate a string in the Python shell without the **print** function, you can see the literal for the **newline character**, **\n**, embedded in the result, as follows:

```
>>> """This very long sentence extends all the way to
the next line."""
'This very long sentence extends all the way to\nthe next line.'
>>>
```

2.3.3 Escape Sequences

The newline character `\n` is called an **escape sequence**. Escape sequences are the way Python expresses special characters, such as the tab, the newline, and the backspace (delete key), as literals. Table 2.3 lists some escape sequences in Python.

ESCAPE SEQUENCE	MEANING
<code>\b</code>	Backspace
<code>\n</code>	Newline
<code>\t</code>	Horizontal tab
<code>\\</code>	The <code>\</code> character
<code>\'</code>	Single quotation mark
<code>\"</code>	Double quotation mark

[TABLE 2.3] Some escape sequences in Python

Because the backslash is used for escape sequences, it must be escaped to appear as a literal character in a string. Thus, `print("\\")` would display a single `\` character.

2.3.4 String Concatenation

You can join two or more strings to form a new string using the concatenation operator `+`. Here is an example:

```
>>> "Hi " + "there, " + "Ken!"
'Hi there, Ken!'
>>>
```

The `*` operator allows you to build a string by repeating another string a given number of times. The left operand is a string, and the right operand is an integer. For example, if you want the string **"Python"** to be preceded by 10 spaces, it would be easier to use the `*` operator with 10 and one space than to enter the 10 spaces by hand. The next session shows the use of the `*` and `+` operators to achieve this result:

```
>>> " " * 10 + "Python"
'          Python'
>>>
```

2.3.5 Variables and the Assignment Statement

As we saw in Chapter 1, a **variable** associates a name with a value, making it easy to remember and use the value later in a program. You need to be mindful of a few rules when choosing names for your variables. For example, some names, such as **if**, **def**, and **import**, are reserved for other purposes and thus cannot be used for variable names. In general, a variable name must begin with either a letter or an underscore (`_`), and can contain any number of letters, digits, or other underscores. Python variable names are case sensitive; thus, the variable **WEIGHT** is a different name from the variable **weight**. Python programmers typically use lowercase letters for variable names, but in the case of variable names that consist of more than one word, it's common to begin each word in the variable name (except for the first one) with an uppercase letter. This makes the variable name easier to read. For example, the name **interestRate** is slightly easier to read than the name **interestrate**.

Programmers use all uppercase letters for the names of variables that contain values that the program never changes. Such variables are known as **symbolic constants**. Examples of symbolic constants in the tax calculator case study are **TAX_RATE** and **STANDARD_DEDUCTION**.

Variables receive their initial values and can be reset to new values with an **assignment statement**. The form of an assignment statement is the following:

```
<variable name> = <expression>
```

The Python interpreter first evaluates the expression on the right side of the assignment symbol and then binds the variable name on the left side to this value. When this happens to the variable name for the first time, it is called **defining** or **initializing** the variable. Note that the **=** symbol means assignment, not equality. After you initialize a variable, subsequent uses of the variable name in expressions are known as **variable references**.

When the interpreter encounters a variable reference in any expression, it looks up the associated value. If a name is not yet bound to a value when it is referenced, Python signals an error. The next session shows some definitions of variables and their references:

```
>>> firstName = "Ken"
>>> secondName = "Lambert"
>>> fullName = firstName + " " + secondName
>>> fullName
'Ken Lambert'
>>>
```

The first two statements initialize the variables **firstName** and **secondName** to string values. The next statement references these variables, concatenates the values referenced by the variables to build a new string, and assigns the result to the variable **fullName**. The last line of code is a simple reference to the variable **fullName**, which returns its value.

Variables serve two important purposes in programs. They help the programmer keep track of data that change over the course of time. They also allow the programmer to refer to a complex piece of information with a simple name. Any time you can substitute a simple thing for a more complex one in a program, you make the program easier for programmers to understand and maintain. Such a process of simplification is called **abstraction**, and it is one of the fundamental ideas of computer science. Throughout this book, you'll learn about other abstractions used in computing, including functions, modules, and classes.

The wise programmer selects names that inform the human reader about the purpose of the data. This, in turn, makes the program easier to maintain and troubleshoot. A good program not only performs its task correctly, but it also reads like an essay in which each word is carefully chosen to convey the appropriate meaning to the reader. For example, a program that creates a payment schedule for a simple interest loan might use the variables **rate**, **initialAmount**, **currentBalance**, and **interest**.

2.3.6 Program Comments and Docstrings

We conclude this subsection on strings with a discussion of **program comments**. A comment is a piece of program text that the interpreter ignores but that provides useful documentation to programmers. At the very least, the author of a program can include his or her name and a brief statement about the purpose of the program at the beginning of the program file. This type of comment, called a **docstring**, is a multi-line string of the form discussed earlier in this section. Here is a docstring that begins a typical program for a lab session:

```
"""
Program: circle.py
Author: Ken Lambert
Last date modified: 2/10/11

The purpose of this program is to compute the area of a circle.
The input is an integer or floating-point number representing the
radius of the circle. The output is a floating-point number
labeled the area of the circle.
"""
```


In addition to docstrings, **end-of-line comments** can document a program. These comments begin with the **#** symbol and extend to the end of a line. An end-of-line comment might explain the purpose of a variable or the strategy used by a piece of code, if it is not already obvious. Here is an example:

```
>>> RATE = 0.85    # Conversion rate for Canadian to US dollars
```

Throughout this book, both types of documentation are colored in green.

Good documentation can be as important in a program as its executable code. Ideally, program code is self-documenting, so a human reader can instantly understand it. However, a program is often read by people who are not its authors, and even the authors might find their own code inscrutable after months of not seeing it. The trick is to avoid documenting code that has an obvious meaning, but to aid the poor reader when the code alone might not provide sufficient understanding. With this end in mind, it's a good idea to do the following:

- 1 Begin a program with a statement of its purpose and other information that would help orient a programmer called on to modify the program at some future date.
- 2 Accompany a variable definition with a comment that explains the variable's purpose.
- 3 Precede major segments of code with brief comments that explain their purpose. The case study program presented earlier in this chapter does this.
- 4 Include comments to explain the workings of complex or tricky sections of code.

2.3 Exercises

- 1 Let the variable **x** be **"dog"** and the variable **y** be **"cat"**. Write the values returned by the following operations:
 - a **x + y**
 - b **"the " + x + " chases the " + y**
 - c **x * 4**
- 2 Write a string that contains your name and address on separate lines using embedded newline characters. Then write the same string literal without the newline characters.

- 3 How does one include an apostrophe as a character within a string literal?
- 4 What happens when the **print** function prints a string literal with embedded newline characters?
- 5 Which of the following are valid variable names?
 - a **length**
 - b **_width**
 - c **firstBase**
 - d **2MoreToGo**
 - e **halt!**
- 6 List two of the purposes of program documentation.

2.4 Numeric Data Types and Character Sets

The first applications of computers were to crunch numbers. Although text and media processing have lately been of increasing importance, the use of numbers in many applications is still very important. In this section, we give a brief overview of numeric data types and their cousins, character sets.

2.4.1 Integers

As you learned in mathematics, the **integers** include 0, all of the positive whole numbers, and all of the negative whole numbers. Integer literals in a Python program are written without commas, and a leading negative sign indicates a negative value.

Although the range of integers is infinite, a real computer's memory places a limit on the magnitude of the largest positive and negative integers. The most common implementation of the **int** data type in many programming languages consists of the integers from $-2,147,483,648$ (-2^{31}) to $2,147,483,647$ ($2^{31} - 1$). However, the magnitude of a long integer can be quite large, but is still limited by the memory of your particular computer. As an experiment, try evaluating the expression **2147483647 ** 100**, which raises the largest positive **int** value to the 100th power. You will see a number that contains many lines of digits!

2.4.2 Floating-Point Numbers

A real number in mathematics, such as the value of pi (3.1416...), consists of a whole number, a decimal point, and a fractional part. Real numbers have **infinite precision**, which means that the digits in the fractional part can continue forever. Like the integers, real numbers also have an infinite range. However, because a computer's memory is not infinitely large, a computer's memory limits not only the range but also the precision that can be represented for real numbers. Python uses **floating-point** numbers to represent real numbers. Values of the most common implementation of Python's **float** type range from approximately -10^{308} to 10^{308} and have 16 digits of precision.

A floating-point number can be written using either ordinary **decimal notation** or **scientific notation**. Scientific notation is often useful for mentioning very large numbers. Table 2.4 shows some equivalent values in both notations.

DECIMAL NOTATION	SCIENTIFIC NOTATION	MEANING
3.78	3.78e0	3.78×10^0
37.8	3.78e1	3.78×10^1
3780.0	3.78e3	3.78×10^3
0.378	3.78e-1	3.78×10^{-1}
0.00378	3.78e-3	3.78×10^{-3}

[TABLE 2.4] Decimal and scientific notations for floating-point numbers

2.4.3 Character Sets

Some programming languages use different data types for strings and individual characters. In Python, character literals look just like string literals and are of the string type. But they also belong to several different **character sets**, among them the **ASCII set** and the **Unicode set**. (The term ASCII stands for American

Standard Code for Information Interchange.) In the 1960s, the original ASCII set encoded each keyboard character and several control characters using the integers from 0 through 127. An example of a control character is Control+D, which is the command to terminate a shell window. As new function keys and some international characters were added to keyboards, the ASCII set doubled in size to 256 distinct values in the mid-1980s. Then, when characters and symbols were added from languages other than English, the Unicode set was created to support 65,536 values in the early 1990s.

Table 2.5 shows the mapping of character values to the first 128 ASCII codes. The digits in the left column represent the leftmost digits of an ASCII code, and the digits in the top row are the rightmost digits. Thus, the ASCII code of the character **'R'** at row 8, column 2 is 82.

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	"	#	\$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	DEL		

[TABLE 2.5] The original ASCII character set

Some might think it odd to include characters in a discussion of numeric types. However, as you can see, the ASCII character set maps to a set of integers. Python's **ord** and **chr** functions convert characters to their numeric ASCII codes and back again, respectively. The next session uses the following functions to explore the ASCII system:

```
>>> ord('a')
97
>>> ord('A')
65
>>> chr(65)
'A'
>>> chr(66)
'B'
>>>
```

Note that the ASCII code for **'B'** is the next number in the sequence after the code for **'A'**. These two functions provide a handy way to shift letters by a fixed amount. For example, if you want to shift three places to the right of the letter **'A'**, you can write **chr(ord('A') + 3)**.

2.4 Exercises

- 1 Which data type would most appropriately be used to represent the following data values?
 - a The number of months in a year
 - b The area of a circle
 - c The current minimum wage
 - d The approximate age of the universe (12,000,000,000 years)
 - e Your name
- 2 Explain the differences between the data types **int** and **float**.
- 3 Write the values of the following floating-point numbers in Python's scientific notation:
 - a 355.76
 - b 0.007832
 - c 4.3212
- 4 Consult Table 2.5 to write the ASCII values of the characters **'\$'** and **'&'**.

2.5 Expressions

As we have seen, a literal evaluates to itself, whereas a variable reference evaluates to the variable's current value. **Expressions** provide an easy way to perform operations on data values to produce other data values. When entered at the Python shell prompt, an expression's operands are evaluated, and its operator is then applied to these values to compute the value of the expression. In this section, we examine arithmetic expressions in more detail.

2.5.1 Arithmetic Expressions

An **arithmetic expression** consists of operands and operators combined in a manner that is already familiar to you from learning algebra. Table 2.6 shows several arithmetic operators and gives examples of how you might use them in Python code.

OPERATOR	MEANING	SYNTAX
-	Negation	-a
**	Exponentiation	a ** b
*	Multiplication	a * b
/	Division	a / b
//	Quotient	a // b
%	Remainder or modulus	a % b
+	Addition	a + b
-	Subtraction	a - b

[TABLE 2.6] Arithmetic operators

In algebra, you are probably used to indicating multiplication like this: **ab**. However, in Python, we must indicate multiplication explicitly, using the multiplication operator (*****), like this: **a * b**. Binary operators are placed between their operands (**a * b**, for example), whereas unary operators are placed before their operands (**-a**, for example).

The **precedence rules** you learned in algebra apply during the evaluation of arithmetic expressions in Python:

- Exponentiation has the highest precedence and is evaluated first.
- Unary negation is evaluated next, before multiplication, division, and remainder.

- Multiplication, both types of division, and remainder are evaluated before addition and subtraction.
- Addition and subtraction are evaluated before assignment.
- With two exceptions, operations of equal precedence are **left associative**, so they are evaluated from left to right. Exponentiation and assignment operations are **right associative**, so consecutive instances of these are evaluated from right to left.
- You can use parentheses to change the order of evaluation.

Table 2.7 shows some arithmetic expressions and their values.

EXPRESSION	EVALUATION	VALUE
$5 + 3 * 2$	$5 + 6$	11
$(5 + 3) * 2$	$8 * 2$	16
$6 \% 2$	0	0
$2 * 3 ** 2$	$2 * 9$	18
$-3 ** 2$	$-(3 ** 2)$	-9
$(3) ** 2$	9	9
$2 ** 3 ** 2$	$2 ** 9$	512
$(2 ** 3) ** 2$	$8 ** 2$	64
$45 / 0$	Error: cannot divide by 0	
$45 \% 0$	Error: cannot divide by 0	

[TABLE 2.7] Some arithmetic expressions and their values

The last two lines of Table 2.7 show attempts to divide by 0, which result in an error. These expressions are good illustrations of the difference between syntax and **semantics**. Syntax is the set of rules for constructing well-formed expressions or sentences in a language. Semantics is the set of rules that allow an agent to interpret the meaning of those expressions or sentences. A computer generates a syntax error when an expression or sentence is not well formed. A **semantic error** is detected when the action that an expression describes cannot be carried out, even though that expression is syntactically correct. Although the expressions $45 / 0$ and $45 \% 0$ are syntactically correct, they are meaningless, because a computing agent cannot carry them out. Human beings can tolerate all kinds of syntax errors and semantic errors when they converse in natural languages. By contrast, computing agents can tolerate none of these errors.

With the exception of exact division, when both operands of an arithmetic expression are of the same numeric type (**int**, **long**, or **float**), the resulting value is also of that type. When each operand is of a different type, the resulting value is of the more general type. Note that the **float** type is more general than the **int** type. The quotient operator `//` produces an integer quotient, whereas the exact division operator `/` always produces a float. Thus, `3 // 4` produces `0`, whereas `3 / 4` produces `.75`.

Although spacing within an expression is not important to the Python interpreter, programmers usually insert a single space before and after each operator to make the code easier for people to read. Normally, an expression must be completed on a single line of Python code. When an expression becomes long or complex, you can move to a new line by placing a backslash character `\` at the end of the current line. The next example shows this technique:

```
>>> 3 + 4 * \
2 ** 5
131
>>>
```

Make sure to insert the backslash before or after an operator. If you break lines in this manner in IDLE, the editor automatically indents the code properly.

As you will see shortly, you can also break a long line of code immediately after a comma. Examples include function calls with several arguments.

2.5.2 Mixed-Mode Arithmetic and Type Conversions

When working with a handheld calculator, we do not give much thought to the fact that we intermix integers and floating-point numbers. Performing calculations involving both integers and floating-point numbers is called **mixed-mode arithmetic**. For instance, if a circle has radius 3, we compute the area as follows:

```
>>> 3.14 * 3 ** 2
28.26
```

How do we perform a similar calculation in Python? In a binary operation on operands of different numeric types, the less general type (**int**) is temporarily and automatically converted to the more general type (**float**) before the operation is performed. Thus, in the example expression, the value 9 is converted to 9.0 before the multiplication.

Remember that Python has different operators for quotient and exact division. For instance,

```
3 // 2 * 5.0 yields 1 * 5.0, which yields 5.0
```

whereas

```
3 / 2 * 5 yields 1.5 * 5, which yields 7.5
```

In general, when you want the most precise results, you should use exact division.

You must use a **type conversion function** when working with the input of numbers. A type conversion function is a function with the same name as the data type to which it converts. Because the **input** function returns a string as its value, you must use the function **int** or **float** to convert the string to a number before performing arithmetic, as in the following example:

```
>>> radius = input("Enter the radius: ")
Enter the radius: 3.2
>>> radius
'3.2'
>>> float(radius)
3.2
>>> float(radius) ** 2 * 3.14
32.153600000000004
```

Table 2.8 lists some common type conversion functions and their uses.

CONVERSION FUNCTION	EXAMPLE USE	VALUE RETURNED
<code>int(<a number or a string>)</code>	<code>int(3.77)</code>	3
	<code>int("33")</code>	33
<code>float(<a number or a string>)</code>	<code>float(22)</code>	22.0
<code>str(<any value>)</code>	<code>str(99)</code>	'99'

[TABLE 2.8] Type conversion functions

Note that the **int** function converts a **float** to an **int** by truncation, not by rounding to the nearest whole number. Truncation simply chops off the number's

fractional part. The **round** function rounds a **float** to the nearest **int** as in the next example:

```
>>> int(6.75)
6
>>> round(6.75)
7
```

Another use of type conversion occurs in the construction of strings from numbers and other strings. For instance, assume that the variable **profit** refers to a floating-point number that represents an amount of money in dollars and cents. Suppose that, to build a string that represents this value for output, we need to concatenate the **\$** symbol to the value of **profit**. However, Python does not allow the use of the **+** operator with a string and a number:

```
>>> profit = 1000.55
>>> print('$' + profit)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'float' objects
```

To solve this problem, we use the **str** function to convert the value of **profit** to a string and then concatenate this string to the **\$** symbol, as follows:

```
>>> print('$' + str(profit))
$1000.55
```

Python is a **strongly typed programming language**. The interpreter checks data types of all operands before operators are applied to those operands. If the type of an operand is not appropriate, the interpreter halts execution with an error message. This error checking prevents a program from attempting to do something that it cannot do.

2.5 Exercises

- 1 Let **x** = 8 and **y** = 2. Write the values of the following expressions:
 - a **x + y * 3**
 - b **(x + y) * 3**
 - c **x ** y**
 - d **x % y**
 - e **x / 12.0**
 - f **x // 6**
- 2 Let **x** = 4.66. Write the values of the following expressions:
 - a **round(x)**
 - b **int(x)**
- 3 How does a Python programmer round a **float** value to the nearest **int** value?
- 4 How does a Python programmer concatenate a numeric value to a string value?
- 5 Assume that the variable **x** has the value 55. Use an assignment statement to increment the value of **x** by 1.

2.6 Using Functions and Modules

Thus far in this chapter, we have examined two ways to manipulate data within expressions. We can apply an operator such as **+** to one or more operands to produce a new data value. Alternatively, we can call a function such as **round** with one or more data values to produce a new data value. Python includes many useful functions, which are organized in libraries of code called **modules**. In this section, we examine the use of functions and modules.

2.6.1 Calling Functions: Arguments and Return Values

A **function** is a chunk of code that can be called by name to perform a task. Functions often require **arguments**, that is, specific data values, to perform their tasks. Arguments are also known as **parameters**. When a function completes its task (which is usually some kind of computation), the function may send a result back to the part of the program that called that function in the first place. The process of sending a result back to another part of a program is known as **returning a value**.

For example, the argument in the function call `round(6.5)` is the value **6.5**, and the value returned is **7**. When an argument is an expression, it is first evaluated, and then its value is passed to the function for further processing. For instance, the function call `abs(4 - 5)` first evaluates the expression `4 - 5` and then passes the result, **-1**, to `abs`. Finally, `abs` returns **1**.

The values returned by function calls can be used in expressions and statements. For example, the function call `print(abs(4 - 5) + 3)` prints the value **4**.

Some functions have only **optional arguments**, some have **required arguments**, and some have both required and optional arguments. For example, the `round` function has one required argument, the number to be rounded. When called with just one argument, the `round` function exhibits its **default behavior**, which is to return the nearest **float** with a fractional part of 0. However, when a second, optional argument is supplied, this argument, a number, indicates the number of places of precision to which the first argument should be rounded. For example, `round(7.563, 2)` returns **7.56**.

To learn how to use a function's arguments, consult the documentation on functions in the shell. For example, Python's `help` function displays information about `round`, as follows:

```
>>> help(round)

Help on built-in function round in module builtin:

round(...)
    round(number[, ndigits]) -> floating point number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the same type as
    number. ndigits may be negative.
```

Each argument passed to a function has a specific data type. When writing code that involves functions and their arguments, you need to keep these data

types in mind. A program that attempts to pass an argument of the wrong data type to a function will usually generate an error. For example, one cannot take the square root of a string, but only of a number. Likewise, if a function call is placed in an expression that expects a different type of operand than that returned by the function, an error will be raised. If you're not sure of the data type associated with a particular function's arguments, read the documentation.

2.6.2 The `math` Module

Functions and other resources are coded in components called **modules**. Functions like **abs** and **round** from the `__builtin__` module are always available for use, whereas the programmer must explicitly import other functions from the modules where they are defined.

The **math** module includes several functions that perform basic mathematical operations. The next code session imports the **math** module and lists a directory of its resources:

```
>>> import math
>>> dir(math)
['_doc_', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',
'exp', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'hypot',
'isinf', 'isnan', 'ldexp', 'log', 'log10', 'loglp', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

This list of function names includes some familiar trigonometric functions as well as Python's most exact estimates of the constants π and **e**.

To use a resource from a module, you write the name of a module as a qualifier, followed by a dot (.) and the name of the resource. For example, to use the value of **pi** from the **math** module, you would write the following code: **math.pi**. The next session uses this technique to display the value of π and the square root of 2:

```
>>> math.pi
3.1415926535897931
>>> math.sqrt(2)
1.4142135623730951
```

Once again, help is available if needed:

```
>>> help(math.cos)

Help on built-in function cos in module math:

cos(...)
    cos(x)

    Return the cosine of x (measured in radians).
```

Alternatively, you can browse through the documentation for the entire module by entering **help(math)**. The function **help** uses a module's own docstring and the docstrings of all its functions to print the documentation.

If you are going to use only a couple of a module's resources frequently, you can avoid the use of the qualifier with each reference by importing the individual resources, as follows:

```
>>> from math import pi, sqrt
>>> print(pi, sqrt(2))
3.14159265359 1.41421356237
>>>
```

Programmers occasionally import all of a module's resources to use without the qualifier. For example, the statement **from math import *** would import all of the **math** module's resources.

Generally, the first technique of importing resources (that is, importing just the module's name) is preferred. The use of a module qualifier not only reminds the reader of a function's purpose, but also helps the interpreter to discriminate between different functions that have the same name.

2.6.3 The Main Module

In the case study, earlier in this chapter, we showed how to write documentation for a Python script. To differentiate this script from the other modules in a program (and there could be many), we call it the **main module**. Like any module, the main module can also be imported. Instead of launching the script from a terminal prompt or loading it into the shell from IDLE, you can start Python from

the terminal prompt and import the script as a module. Let's do that with the **taxform.py** script, as follows:

```
>>> import taxform
Enter the gross income: 120000
Enter the number of dependents: 2
The income tax is $20800.0
```

After importing a main module, you can view its documentation by running the **help** function:

```
>>> help(taxform)

DESCRIPTION
  Program: taxform.py
  Author: Ken

  Compute a person's income tax.

  1. Significant constants
      tax rate
      standard deduction
      deduction per dependent
  2. The inputs are
      gross income
      number of dependents
  3. Computations:
      net income = gross income - the standard deduction -
                    a deduction for each dependent
      income tax = is a fixed percentage of the net income
  4. The outputs are
      the income tax
```

2.6.4 Program Format and Structure

This is a good time to step back and get a sense of the overall format and structure of simple Python programs. It's a good idea to structure your programs as follows:

- Start with an introductory comment stating the author's name, the purpose of the program, and other relevant information. This information should be in the form of a docstring.

- Then, include statements that do the following:
 - Import any modules needed by the program.
 - Initialize important variables, suitably commented.
 - Prompt the user for input data and save the input data in variables.
 - Process the inputs to produce the results.
 - Display the results.

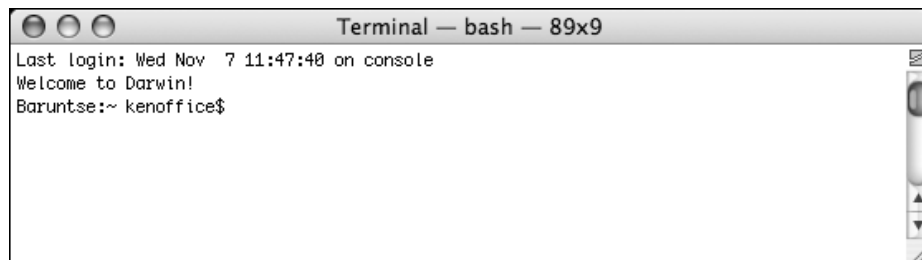
Take a moment to review the income tax program presented in the case study at the beginning of this chapter. Notice how the program conforms to this basic organization. Also, notice that the various sections of the program are separated by whitespace (blank lines). Remember, programs should be easy for other programmers to read and understand. They should read like essays!

2.6.5

Running a Script from a Terminal Command Prompt

Thus far in this book, we have been developing and running Python programs experimentally in IDLE. When a program's development and testing are finished, the program can be released to others to run on their computers. Python must be installed on a user's computer, but the user need not run IDLE to run a Python script.

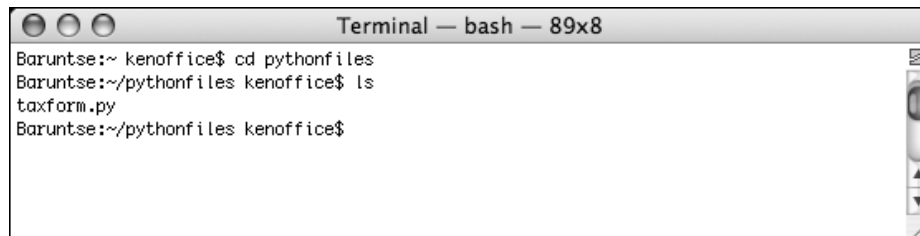
One way to run a Python script is to open a terminal command prompt window. On a computer running Windows, this is the DOS command prompt window; to open it, select the **Start** button, select **All Programs**, select **Accessories**, and then select **Command Prompt**. On a Macintosh or UNIX-based system, this is a terminal window. A terminal window on a Macintosh is shown in Figure 2.5.



[FIGURE 2.5] A terminal window on a Macintosh

After the user has opened a terminal window, she must navigate or change directories until the prompt shows that she is attached to the directory that contains the Python script. For example, if we assume that the script named **taxform.py** is in

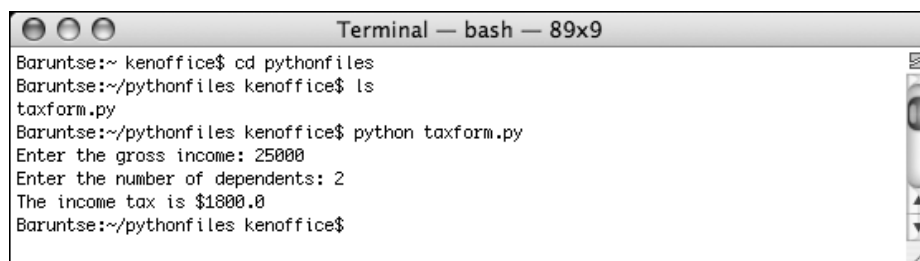
the **pythonfiles** directory under the terminal's current directory, Figure 2.6 shows the commands to change to this directory and list its contents.

A terminal window titled "Terminal — bash — 89x8" showing a user named Baruntse at a machine named kenoffice. The user enters the command `cd pythonfiles` to change the current directory. Then, they enter `ls` to list the contents of the current directory, which outputs `taxform.py`.

```
Baruntse:~ kenoffice$ cd pythonfiles
Baruntse:~/pythonfiles kenoffice$ ls
taxform.py
Baruntse:~/pythonfiles kenoffice$
```

[FIGURE 2.6] Changing to another directory and listing its contents

When the user is attached to the appropriate directory, she can run the script by entering the command **python scriptname.py** at the command prompt. Figure 2.7 shows this step and a run of the **taxform** script.

A terminal window titled "Terminal — bash — 89x9" showing the same user and machine. The user enters `cd pythonfiles` and `ls`. Then, they enter `python taxform.py`. The script prompts for "Enter the gross income: 25000" and "Enter the number of dependents: 2". It then outputs "The income tax is \$1800.0".

```
Baruntse:~ kenoffice$ cd pythonfiles
Baruntse:~/pythonfiles kenoffice$ ls
taxform.py
Baruntse:~/pythonfiles kenoffice$ python taxform.py
Enter the gross income: 25000
Enter the number of dependents: 2
The income tax is $1800.0
Baruntse:~/pythonfiles kenoffice$
```

[FIGURE 2.7] Running a Python script in a terminal window

All Python installations also provide the capability of launching Python scripts by double-clicking the files from the operating system's file browser. On Windows systems, this feature is automatic, whereas on Macintosh and UNIX-based systems, the **.py** file type must be set to launch with the Python launcher application. When you launch a script in this manner, however, the command prompt window opens, shows the output of the script, and closes. To prevent this fly-by-window problem, you can add an input statement at the end of the script that pauses until the user presses the enter or return key, as follows:

```
input("Please press enter or return to quit the program. ")
```

2.6

Exercises

- 1 Explain the relationship between a function and its arguments.
- 2 The **math** module includes a **pow** function that raises a number to a given power. The first argument is the number, and the second argument is the exponent. Write a code segment that imports this function and calls it to print the values 8^2 and 5^4 .
- 3 Explain how to display a directory of all of the functions in a given module.
- 4 Explain how to display help information on a particular function in a given module.

Summary

- The waterfall model describes the software development process in terms of several phases. Analysis determines what the software will do. Design determines how the software will accomplish its purposes. Implementation involves coding the software in a particular programming language. Testing and integration demonstrate that the software does what it is intended to do as it is put together for release. Maintenance locates and fixes errors after release and adds new features to the software.
- Literals are data values that can appear in a program. They evaluate to themselves.
- The string data type is used to represent text for input and output. Strings are sequences of characters. String literals are enclosed in pairs of single or double quotation marks. Two strings can be combined by concatenation to form a new string.
- Escape characters begin with a backslash and represent special characters such as the delete key and the newline.
- A docstring is a string enclosed by triple quotation marks and provides program documentation.
- Comments are pieces of code that are not evaluated by the interpreter but can be read by programmers to obtain information about a program.

- Variables are names that refer to values. The value of a variable is initialized and can be reset by an assignment statement. In Python, any variable can name any value.
- The **int** data type represents integers. The **float** data type represents floating-point numbers. The magnitude of an integer or a floating-point number is limited by the memory of the computer, as is the number's precision in the case of floating-point numbers.
- Arithmetic operators are used to form arithmetic expressions. Operands can be numeric literals, variables, function calls, or other expressions.
- The operators are ranked in precedence. In descending order, they are exponentiation, negation, multiplication (*, /, and % are the same), addition (+ and – are the same), and assignment. Operators with a higher precedence are evaluated before those with a lower precedence. Normal precedence can be overridden by parentheses.
- Mixed-mode operations involve operands of different numeric data types. They result in a value of the more inclusive data type.
- The type conversion functions can be used to convert a value of one type to a value of another type after input.
- A function call consists of a function's name and its arguments or parameters. When it is called, the function's arguments are evaluated, and these values are passed to the function's code for processing. When the function completes its work, it may return a result value to the caller.
- Python is a strongly typed language. The interpreter checks the types of all operands within expressions and halts execution with an error if they are not as expected for the given operators.
- A module is a set of resources, such as function definitions. Programmers access these resources by importing them from their modules.
- A semantic error occurs when the computer cannot perform the requested operation, such as an attempt to divide by 0. Python programs with semantic errors halt with an error message.
- A logic error occurs when a program runs to a normal termination but produces incorrect results.

REVIEW QUESTIONS

- 1 What does a programmer do during the analysis phase of software development?
 - a Codes the program in a particular programming language
 - b Writes the algorithms for solving a problem
 - c Decides what the program will do and determines its user interface
 - d Tests the program to verify its correctness
- 2 What must a programmer use to test a program?
 - a All possible sets of legitimate inputs
 - b All possible sets of inputs
 - c A single set of legitimate inputs
 - d A reasonable set of legitimate inputs
- 3 What must you use to create a multi-line string?
 - a A single pair of double quotation marks
 - b A single pair of single quotation marks
 - c A single pair of three consecutive double quotation marks
 - d Embedded newline characters
- 4 What is used to begin an end-of-line comment?
 - a / symbol
 - b # symbol
 - c % symbol
- 5 Which of the following lists of operators is ordered by decreasing precedence?
 - a +, *, **
 - b *, /, %
 - c **, *, +
- 6 The expression `2 ** 3 ** 2` evaluates to which of the following values?
 - a 64
 - b 512
 - c 8

- 7 The expression `round(23.67)` evaluates to which of the following values?
 - a 23
 - b 23.7
 - c 24.0
- 8 Assume that the variable `name` has the value 33. What is the value of `name` after the assignment statement `name = name * 2` executes?
 - a 35
 - b 33
 - c 66
- 9 Write an import statement that imports just the functions `sqrt` and `log` from the `math` module.
- 10 What is the purpose of the `dir` function and the `help` function?

PROJECTS

In each of the projects that follow, you should write a program that contains an introductory docstring. This documentation should describe what the program will do (analysis) and how it will do it (design the program in the form of a pseudocode algorithm). Include suitable prompts for all inputs, and label all outputs appropriately. After you have coded a program, be sure to test it with a reasonable set of legitimate inputs.

- 1 The tax calculator program of the case study outputs a floating-point number that might show more than two digits of precision. Use the `round` function to modify the program to display at most two digits of precision in the output number.
- 2 You can calculate the surface area of a cube if you know the length of an edge. Write a program that takes the length of an edge (an integer) as input and prints the cube's surface area as output.

- 3 Five Star Video rents new videos for \$3.00 a night, and oldies for \$2.00 a night. Write a program that the clerks at Five Star Video can use to calculate the total charge for a customer's video rentals. The program should prompt the user for the number of each type of video and output the total cost.
- 4 Write a program that takes the radius of a sphere (a floating-point number) as input and outputs the sphere's diameter, circumference, surface area, and volume.
- 5 An object's momentum is its mass multiplied by its velocity. Write a program that accepts an object's mass (in kilograms) and velocity (in meters per second) as inputs and then outputs its momentum.
- 6 The kinetic energy of a moving object is given by the formula $KE = (1/2)mv^2$, where m is the object's mass and v is its velocity. Modify the program you created in Project 5 so that it prints the object's kinetic energy as well as its momentum.
- 7 Write a program that calculates and prints the number of minutes in a year.
- 8 Light travels at $3 * 10^8$ meters per second. A light-year is the distance a light beam travels in one year. Write a program that calculates and displays the value of a light-year.
- 9 Write a program that takes as input a number of kilometers and prints the corresponding number of nautical miles. Use the following approximations:
 - A kilometer represents 1/10,000 of the distance between the North Pole and the equator.
 - There are 90 degrees, containing 60 minutes of arc each, between the North Pole and the equator.
 - A nautical mile is 1 minute of an arc.
- 10 An employee's total weekly pay equals the hourly wage multiplied by the total number of regular hours plus any overtime pay. Overtime pay equals the total overtime hours multiplied by 1.5 times the hourly wage. Write a program that takes as inputs the hourly wage, total regular hours, and total overtime hours and displays an employee's total weekly pay.

[CHAPTER] 3 Control Statements

After completing this chapter, you will be able to:

- Write a loop to repeat a sequence of actions a fixed number of times
- Write a loop to traverse the sequence of characters in a string
- Write a loop that counts down and a loop that counts up
- Write an entry-controlled loop that halts when a condition becomes false
- Use selection statements to make choices in a program
- Construct appropriate conditions for condition-controlled loops and selection statements
- Use logical operators to construct compound Boolean expressions
- Use a selection statement and a break statement to exit a loop that is not entry-controlled

All the programs you have studied so far in this book have consisted of short sequences of instructions that are executed one after the other. Even if we allowed the sequence of instructions to be quite long, this type of program would not be very useful. Like human beings, computers must be able to repeat a set of actions. They also must be able to select an action to perform in a particular situation. This chapter focuses on **control statements**—statements that allow the computer to select or repeat an action.

3.1 Definite Iteration: The `for` Loop

We begin our study of control statements with repetition statements, also known as **loops**, which repeat an action. Each repetition of the action is known as a **pass** or an **iteration**. There are two types of loops—those that repeat an action a pre-defined number of times (**definite iteration**) and those that perform the action until the program determines that it needs to stop (**indefinite iteration**). In this section, we examine Python's **for loop**, the control statement that most easily supports definite iteration.

3.1.1 Executing a Statement a Given Number of Times

When Dr. Frankenstein's monster came to life, the good doctor exclaimed, "It's alive! It's alive!" A computer can easily print these exclamations not just twice, but a dozen or a hundred times. Here is a **for** loop that does so four times:

```
>>> for eachPass in range(4):
    print("It's alive!", end=" ")

It's alive! It's alive! It's alive! It's alive!
>>>
```

This loop repeatedly calls one function—the **print** function. The constant 4 on the first line tells the loop how many times to call this function. If we want to print 10 or 100 exclamations, we just change the 4 to 10 or to 100. The form of this type of loop is

```
for <variable> in range(<an integer expression>):
    <statement-1>

    <statement-n>
```

The first line of code in a loop is sometimes called the **loop header**. For now, the only relevant information in the header is the integer expression, which denotes the number of iterations that the loop performs. The colon (`:`) ends the loop header. The **loop body** comprises the statements in the remaining lines of code, below the header. Note that the statements in the loop body *must be indented and aligned in the same column*. These statements are executed in sequence on each pass through the loop.

Now let's explore how Python's exponentiation operator might be implemented in a loop. Recall that this operator raises a number to a given power. For instance, the expression `2 ** 3` computes the value of 2^3 , or `2 * 2 * 2`. The following session uses a loop to compute an exponentiation for a non-negative exponent. We use three variables to designate the number, the exponent, and the product. The product is initially 1. On each pass through the loop, the product is multiplied by the number and reset to the result. To allow us to trace this process, the value of the product is also printed on each pass.

```
>>> number = 2
>>> exponent = 3
>>> product = 1
>>> for eachPass in range(exponent):
    product = product * number
    print(product, end = " ")

2 4 8
>>> product
8
```

As you can see, if the exponent were 0, the loop body would not execute, and the value of **product** would remain as 1, which is the value of any number raised to the zero power.

The use of variables in the preceding example demonstrates that our exponentiation loop is an algorithm that solves a *general class* of problems. The user of this particular loop not only can raise 2 to the 3rd power, but also can raise any number to any non-negative power, just by substituting different values for the variables **number** and **exponent**.

3.1.2 Count-Controlled Loops

When Python executes the type of **for** loop just discussed, it actually counts from 0 to the value of the header's integer expression minus 1. On each pass

through the loop, the header's variable is bound to the current value of this count. The next code segment demonstrates this fact:

```
>>> for count in range(4):
    print(count, end = " ")

0 1 2 3
>>>
```

Loops that count through a range of numbers are also called **count-controlled loops**. The value of the count on each pass is often used in computations. For example, consider the factorial of 4, which is $1 * 2 * 3 * 4 = 24$. A code segment to compute this value starts with a product of 1 and resets this variable to the result of multiplying it and the loop's count plus 1 on each pass, as follows:

```
>>> product = 1
>>> for count in range(4):
    product = product * (count + 1)

>>> product
24
```

Note that the value of **count + 1** is used on each pass, to ensure that the numbers used are 1 through 4 rather than 0 through 3.

To count from an explicit lower bound, the programmer can supply a second integer expression in the loop header. When two arguments are supplied to **range**, the count ranges from the first argument to the second argument minus 1. The next code segment uses this variation to simplify the code in the loop body:

```
>>> product = 1
>>> for count in range(1, 5):
    product = product * count

>>> product
24
>>>
```

The only thing in this version to be careful about is the second argument of **range**, which should specify an integer greater by 1 than the desired upper bound of the count. Here is the form of this version of the **for** loop:

```
for <variable> in range(<lower bound>, <upper bound + 1>):  
    <loop body>
```

Accumulating a single result value from a series of values is a common operation in computing. Here is an example of a **summation**, which accumulates the sum of a sequence of numbers from a lower bound through an upper bound:

```
>>> lower = int(input("Enter the lower bound: "))  
Enter the lower bound: 1  
>>> upper = int(input("Enter the upper bound: "))  
Enter the upper bound: 10  
>>> sum = 0  
>>> for count in range(lower, upper + 1):  
        sum = sum + count  
  
>>> sum  
55  
>>>
```

3.1.3 Augmented Assignment

Expressions such as **x = x + 1** or **x = x + 2** occur so frequently in loops that Python includes abbreviated forms for them. The assignment symbol can be combined with the arithmetic and concatenation operators to provide **augmented assignment operations**. Following are several examples:

```
a = 17  
s = "hi"  
  
a += 3          # Equivalent to a = a + 3  
a -= 3          # Equivalent to a = a - 3  
a *= 3          # Equivalent to a = a * 3  
a /= 3          # Equivalent to a = a / 3  
a %= 3          # Equivalent to a = a % 3  
s += " there"   # Equivalent to s = s + " there"
```

All these examples have the format

```
<variable> <operator>= <expression>
```

which is equivalent to

```
<variable> = <variable> <operator> <expression>
```

Note that there is no space between **<operator>** and **=**. The augmented assignment operations and the standard assignment operation have the same precedence.

3.1.4 Loop Errors: Off-by-One Error

The **for** loop is not only easy to write, but also fairly easy to write correctly. Once we get the syntax correct, we need to be concerned about only one other possible error: the loop fails to perform the expected number of iterations. Because this number is typically off by one, the error is called an **off-by-one error**. For the most part, off-by-one errors result when the programmer incorrectly specifies the upper bound of the loop. The programmer might intend the following loop to count from 1 through 4, but it actually counts from 1 through 3:

```
for count in range(1, 4):    # Count from 1 through 4, we think
    print(count)
```

Note that this is not a syntax error, but rather a logic error. Unlike syntax errors, logic errors are not detected by the Python interpreter, but only by the eyes of a programmer who carefully inspects a program's output.

3.1.5 Traversing the Contents of a Data Sequence

Although we have been using the **for** loop as a simple count-controlled loop, the loop itself actually visits each number in a sequence of numbers generated

by the **range** function. The next code segment shows what these sequences look like:

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(1, 5))
[1, 2, 3, 4]
>>>
```

In this example, the sequence of numbers generated by the function **range** is fed to Python's **list** function, which returns a special type of sequence called a **list**. Strings are also sequences of characters. The values contained in any sequence can be visited by running a **for** loop, as follows:

```
for <variable> in <sequence>:
    <do something with variable>
```

On each pass through the loop, the variable is bound to or assigned the next value in the sequence, starting with the first one and ending with the last one. The following code segment traverses or visits all the elements in two sequences and prints the values contained in them on single lines:

```
>>> for number in [1, 2, 3]:
    print(number, end = " ")

1 2 3
>>> for character in "Hi there!":
    print(character, end = " ")

H i   t h e r e !
>>>
```

3.1.6 Specifying the Steps in the Range

The count-controlled loops we have seen thus far count through consecutive numbers in a series. However, in some programs we might want a loop to skip some numbers, perhaps visiting every other one or every third one. A variant of

Python's **range** function expects a third argument that allows you to nicely skip some numbers. The third argument specifies a **step value**, or the interval between the numbers used in the range, as shown in the examples that follow:

```
>>> list(range(1, 6, 1))    # Same as using two arguments
[1, 2, 3, 4, 5]
>>> list(range(1, 6, 2))    # Use every other number
[1, 3, 5]
>>> list(range(1, 6, 3))    # Use every third number
[1, 4]
>>>
```

Now, suppose you had to compute the sum of the even numbers between 1 and 10. Here is the code that solves this problem:

```
>>> sum = 0
>>> for count in range(2, 11, 2):
        sum += count

>>> sum
30
>>>
```

3.1.7 Loops That Count Down

All of our loops until now have counted up from a lower bound to an upper bound. Once in a while, a problem calls for counting in the opposite direction, from the upper bound down to the lower bound. For example, when the top-10 singles tunes are released, they might be presented in order from lowest (10th) to highest (1st) rank. In the next session, a loop displays the count from 10 down to 1 to show how this would be done:

```
>>> for count in range(10, 0, -1):
        print(count, end=" ")

10 9 8 7 6 5 4 3 2 1
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

When the step argument is a negative number, the **range** function generates a sequence of numbers from the first argument down to the second argument plus 1. Thus, in this case, the first argument should express the upper bound, and the second argument should express the lower bound minus 1.

3.1 Exercises

- 1 Write the outputs of the following loops:
 - a

```
for count in range(5):  
    print(count + 1, end=" ")
```
 - b

```
for count in range(1, 4):  
    print(count, end=" ")
```
 - c

```
for count in range(1, 6, 2):  
    print(count, end=" ")
```
 - d

```
for count in range(6, 1, -1):  
    print(count, end=" ")
```
- 2 Write a loop that prints your name 100 times. Each output should begin on a new line.
- 3 Explain the role of the variable in the header of a **for** loop.
- 4 Write a loop that prints the first 128 ASCII values followed by the corresponding characters (see the section on characters in Chapter 2).
- 5 Assume that the variable **testString** refers to a string. Write a loop that prints each character in this string, followed by its ASCII value.

3.2 Formatting Text for Output

Before turning to our next case study, we need to examine more closely the format of text for output. Many data-processing applications require output that has a **tabular format**. In this format, numbers and other information are aligned in columns that can be either left-justified or right-justified. A column of data is left-justified if its values are vertically aligned beginning with their leftmost characters. A column of data is right-justified if its values are vertically aligned beginning with their rightmost characters. To maintain the margins between columns

of data, left-justification requires the addition of spaces to the right of the datum, whereas right-justification requires adding spaces to the left of the datum. A column of data is centered if there are an equal number of spaces on either side of the data within that column.

The total number of data characters and additional spaces for a given datum in a formatted string is called its **field width**.

The **print** function automatically begins printing an output datum in the first available column. The next example, which displays the exponents 7 through 10 and the values of 10^7 through 10^{10} , shows the format of two columns produced by the **print** function:

```
>>> for exponent in range(7, 11):
      print(exponent, 10 ** exponent)

7 10000000
8 100000000
9 1000000000
10 10000000000
>>>
```

Note that when the exponent reaches 10, the output of the second column shifts over by a space and looks ragged. The output would look neater if the left column were left-justified and the right column were right-justified. When we format floating-point numbers for output, we often would like to specify the number of digits of precision to be displayed as well as the field width. This is especially important when displaying financial data in which exactly two digits of precision are required.

Python includes a general formatting mechanism that allows the programmer to specify field widths for different types of data. The next session shows how to right-justify and left-justify the string **"four"** within a field width of 6:

```
>>> "%6s" % "four"          # Right justify
'  four'
>>> "%-6s" % "four"         # Left justify
'four  '
```

The first line of code right-justifies the string by padding it with two spaces to its left. The next line of code left-justifies by placing two spaces to the string's right.

The simplest form of this operation is the following:

```
<format string> % <datum>
```

This version contains a **format string**, the **format operator %**, and a single data value to be formatted. The format string can contain string data and other information about the format of the datum. To format the string data value in our example, we used the notation **%<field width>s** in the format string. When the field width is positive, the datum is right-justified; when the field width is negative, you get left-justification. If the field width is less than or equal to the datum's print length in characters, no justification is added. The **%** operator works with this information to build and return a formatted string.

To format integers, you use the letter **d** instead of **s**. To format a sequence of data values, you construct a format string that includes a format code for each datum and place the data values in a tuple following the **%** operator. The form of the second version of this operation follows:

```
<format string> % (<datum-1>, ..., <datum-n>)
```

Armed with the format operation, our powers of 10 loop can now display the numbers in nicely aligned columns. The first column is left-justified in a field width of 3, and the second column is right-justified in a field width of 12.

```
>>> for exponent in range(7, 11):
      print("%-3d%12d" % (exponent, 10 ** exponent))

7      10000000
8      100000000
9      1000000000
10     10000000000
```

The format information for a data value of type **float** has the form:

```
%<field width>.<precision>f
```

where **.<precision>** is optional. The next session shows the output of a floating-point number without, and then with, a format string:

```
>>> salary = 100.00
>>> print("Your salary is $" + str(salary))
Your salary is $100.0
>>> print("Your salary is $%0.2f" % salary)
Your salary is $100.00
>>>
```

Here is another, minimal, example of the use of a format string, which says to use a field width of 6 and a precision of 3 to format the **float** value 3.14:

```
>>> "%6.3f" % 3.14
' 3.140'
```

Note that Python adds a digit of precision to the number's string and pads it with a space to the left to achieve the field width of 6. This width includes the place occupied by the decimal point.

3.2 Exercises

- 1 Assume that the variable **amount** refers to **24.325**. Write the outputs of the following statements:
 - a `print("Your salary is $%0.2f" % amount)`
 - b `print("The area is %0.1f" % amount)`
 - c `print("%7f" % amount)`
- 2 Write a code segment that displays the values of the integers **x**, **y**, and **z** on a single line, such that each value is right-justified in six columns.
- 3 Write a format operation that builds a string for the **float** variable **amount** that has exactly two digits of precision and a field width of zero.
- 4 Write a loop that outputs the numbers in a list named **salaries**. The outputs should be formatted in a column that is right-justified, with a field width of 12 and a precision of 2.

3.3

Case Study: An Investment Report

It has been said that compound interest is the eighth wonder of the world. Our next case study, which computes an investment report, shows why.

3.3.1

Request

Write a program that computes an investment report.

3.3.2

Analysis

The inputs to this program are the following:

- An initial amount to be invested (a floating-point number)
- A period of years (an integer)
- An interest rate (a percentage expressed as an integer)

The program uses a simplified form of compound interest, in which the interest is computed once each year and added to the total amount invested. The output of the program is a report in tabular form that shows, for each year in the term of the investment, the year number, the initial balance in the account, the interest earned for that year, and the ending balance for that year. The columns of the table are suitably labeled with a header in the first row. Following the output of the table, the program prints the total amount of the investment balance and the total amount of interest earned for the period. The proposed user interface is shown in Figure 3-1.

```
Enter the investment amount: 10000.00
Enter the number of years: 5
Enter the rate as a %: 5
Year  Starting balance  Interest  Ending balance
1      10000.00      500.00    10500.00
2      10500.00      525.00    11025.00
3      11025.00      551.25    11576.25
4      11576.25      578.81    12155.06
5      12155.06      607.75    12762.82
Ending balance: $12762.82
Total interest earned: $2762.82
```

[FIGURE 3.1] The user interface for the investment report program

3.3.3 Design

The four principal parts of the program perform the following tasks:

- 1 Receive the user's inputs and initialize data.
- 2 Display the table's header.
- 3 Compute the results for each year, and display them as a row in the table.
- 4 Display the totals.

The third part of the program, which computes and displays the results, is a loop. The following is a slightly simplified version of the pseudocode for the program, without the details related to formatting the outputs:

```
Input the starting balance, number of years, and interest rate
Set the total interest to 0.0
Print the table's heading
For each year
    compute the interest
    compute the ending balance
    print the year, starting balance, interest, and ending balance
    update the starting balance
    update the total interest
print the ending balance and the total interest
```

Ignoring the details of the output at this point allows us to focus on getting the computations correct. We can translate this pseudocode to a Python program to check our computations. A rough draft of a program is called a **prototype**. Once we are confident that the prototype is producing the correct numbers, we can return to the design and work out the details of formatting the outputs.

The format of the outputs is guided by the requirement that they be aligned nicely in columns. We use a format string to right-justify all of the numbers on each row of output. We also use a format string for the string labels in the table's header. After some trial and error, we come up with field widths of 4, 18, 10, and 16 for the year, starting balance, interest, and ending balance, respectively. We can also use these widths in the format string for the header.

3.3.4 Implementation (Coding)

The code for this program shows each of the major parts described in the design, set off by end-of-line comments. Note the use of the many variables to track the

various amounts of money used by the program. Wisely, we have chosen names for these variables that clearly describe their purpose. The format strings in the **print** statements are rather complex, but we have made an effort to format them so the information they contain is still fairly readable.

```
"""
Program: investment.py
Author: Ken

Compute an investment report.

1. The inputs are
    starting investment amount
    number of years
    interest rate (an integer percent)

2. The report is displayed in tabular form with a header.

3. Computations and outputs:
    for each year
        compute the interest and add it to the investment
        print a formatted row of results for that year

4. The ending investment and interest earned are also displayed.
"""

# Accept the inputs
startBalance = float(input("Enter the investment amount: "))
years = int(input("Enter the number of years: "))
rate = int(input("Enter the rate as a %: "))

# Convert the rate to a decimal number
rate = rate / 100

# Initialize the accumulator for the interest
totalInterest = 0.0

# Display the header for the table
print("%4s%18s%10s%16s" % \
      ("Year", "Starting balance",
       "Interest", "Ending balance"))
# Compute and display the results for each year
for year in range(1, years + 1):
    interest = startBalance * rate
    endBalance = startBalance + interest
    print("%4d%18.2f%10.2f%16.2f" % \
          (year, startBalance, interest, endBalance))
```

continued

```

startBalance = endBalance
totalInterest += interest

# Display the totals for the period
print("Ending balance: $%0.2f" % endBalance)
print("Total interest earned: $%0.2f" % totalInterest)

```

3.3.5 Testing

When testing a program that contains a loop, we should focus first on the input that determines the number of iterations. In our program, this value is the number of years. We enter a value that yields the smallest possible number of iterations, then increase this number by 1, then use a slightly larger number, such as 5, and finally we use a number close to the maximum expected, such as 50 (in our problem domain, probably the largest realistic period of an investment). The values of the other inputs, such as the investment amount and the rate in our program, should be reasonably small and stay fixed for this phase of the testing. If the program produces correct outputs for all of these inputs, we can be confident that the loop is working correctly.

In the next phase of testing, we examine the effects of the other inputs on the results, including their format. We know that the other two inputs to our programs, the investment and the rate, already produce correct results for small values. A reasonable strategy might be to test a large investment amount with the smallest and largest number of years and a small rate, and then with the largest number of years and the largest reasonable rate. Table 3-1 organizes these sets of test data for the program.

INVESTMENT	YEARS	RATE
100.00	1	5
100.00	2	5
100.00	5	5
100.00	50	5
10000.00	1	5
10000.00	50	5
10000.00	50	20

[TABLE 3.1] The data sets for testing the investment program

3.4

Selection: `if` and `if-else` Statements

We have seen that computers can plow through long sequences of instructions and that they can do so repeatedly. However, not all problems can be solved in this manner. In some cases, instead of moving straight ahead to execute the next instruction, the computer might be faced with two alternative courses of action. The computer must pause to examine or test a **condition**, which expresses a hypothesis about the state of its world at that point in time. If the condition is true, the computer executes the first alternative action and skips the second alternative. If the condition is false, the computer skips the first alternative action and executes the second alternative. In other words, instead of moving blindly ahead, the computer exercises some intelligence by responding to conditions in its environment. In this section, we explore several types of **selection statements**, or control statements, that allow a computer to make choices. But first, we need to examine how a computer can test conditions.

3.4.1

The Boolean Type, Comparisons, and Boolean Expressions

Before you can test conditions in a Python program, you need to understand the **Boolean data type**, which is named for the nineteenth century British mathematician George Boole. The Boolean data type consists of only two data values—true and false. In Python, Boolean literals can be written in several ways, but most programmers prefer the use of the standard values **True** and **False**.

Simple Boolean expressions consist of the Boolean values **True** or **False**, variables bound to those values, function calls that return Boolean values, or comparisons. The condition in a selection statement often takes the form of a comparison. For example, you might compare value A to value B to see which one is greater. The result of the comparison is a Boolean value. It is either true or false that value A is greater than value B. To write expressions that make comparisons, you have to be familiar with Python's comparison operators, which are listed in Table 3-2.

COMPARISON OPERATOR	MEANING
==	Equals
!=	Not equals
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

[TABLE 3.2] The comparison operators

The following session shows some example comparisons and their values:

```
>>> 4 == 4
True
>>> 4 != 4
False
>>> 4 < 5
True
>>> 4 >= 3
True
>>> "A" < "B"
True
>>>
```

Note that **==** means equals, whereas **=** means assignment. As you learned in Chapter 2, when evaluating expressions in Python, you need to be aware of precedence—that is, the order in which operators are applied in complex expressions. The comparison operators are applied after addition but before assignment.

3.4.2 if-else Statements

The **if-else statement** is the most common type of selection statement. It is also called a **two-way selection statement**, because it directs the computer to make a choice between two alternative courses of action.

The **if-else** statement is often used to check inputs for errors and to respond with error messages if necessary. The alternative is to go ahead and perform the computation if the inputs are valid.

For example, suppose a program inputs the area of a circle and computes and outputs its radius. Legitimate inputs for this program would be positive numbers. But, by mistake, the user could still enter a zero or a negative number. Because the program has no choice but to use this value to compute the radius, it might crash (stop running) or produce a meaningless output. The next code segment shows how to use an **if-else** statement to locate (trap) this error and respond to it:

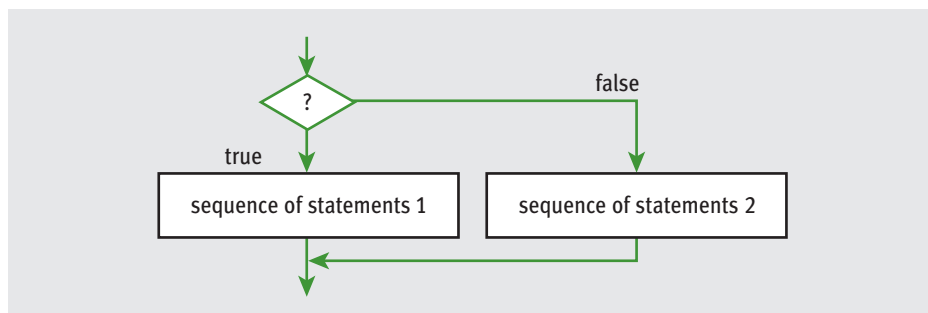
```
import math

area = float(input("Enter the area: "))
if area > 0:
    radius = math.sqrt(area / math.pi)
    print("The radius is", radius)
else:
    print("Error: the area must be a positive number")
```

Here is the Python syntax for the **if-else** statement:

```
if <condition>:
    <sequence of statements-1>
else:
    <sequence of statements-2>
```

The condition in the **if-else** statement must be a Boolean expression—that is, an expression that evaluates to either true or false. The two possible actions each consist of a sequence of statements. Note that each sequence *must be indented at least one space* beyond the symbols **if** and **else**. Lastly, note the use of the colon (:) following the condition and the word **else**. Figure 3-2 shows a flow diagram of the semantics of the **if-else** statement. In that diagram, the diamond containing the question mark indicates the condition.



[FIGURE 3.2] The semantics of the **if-else** statement

Our next example prints the maximum and minimum of two input numbers.

```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
if first > second:
    maximum = first
    minimum = second
else:
    maximum = second
    minimum = first
print("Maximum:", maximum)
print("Minimum:", minimum)
```

Python includes two functions, **max** and **min**, that make the **if-else** statement in this example unnecessary. In the following example, the function **max** returns the largest of its arguments, whereas **min** returns the smallest of its arguments:

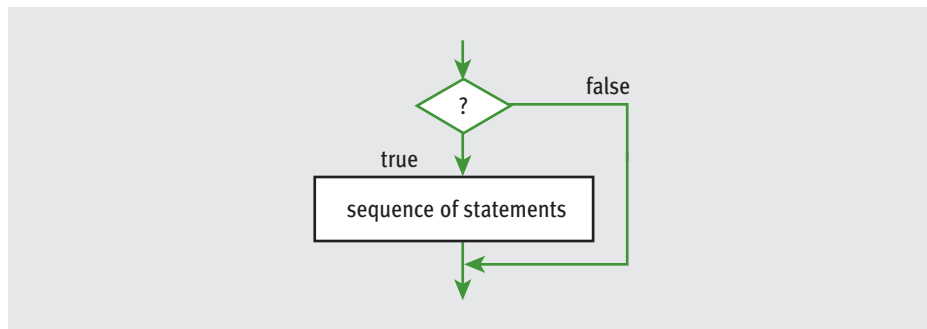
```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
print("Maximum:", max(first, second))
print("Minimum:", min(first, second))
```

3.4.3 One-Way Selection Statements

The simplest form of selection is the **if statement**. This type of control statement is also called a **one-way selection statement**, because it consists of a condition and just a single sequence of statements. If the condition is **True**, the sequence of statements is run. Otherwise, control proceeds to the next statement following the entire selection statement. Here is the syntax for the **if** statement:

```
if <condition>:
    <sequence of statements>
```

Figure 3-3 shows a flow diagram of the semantics of the **if** statement.



[FIGURE 3.3] The semantics of the **if** statement

Simple **if** statements are often used to prevent an action from being performed if a condition is not right. For example, the absolute value of a negative number is the arithmetic negation of that number, otherwise it is just that number. The next session uses a simple **if** statement to reset the value of a variable to its absolute value:

```
>>> if x < 0:
      x = -x
>>>
```

3.4.4 Multi-Way **if** Statements

Occasionally, a program is faced with testing several conditions that entail more than two alternative courses of action. For example, consider the problem of converting numeric grades to letter grades. Table 3-3 shows a simple grading scheme that is based on two assumptions: that numeric grades can range from 0 to 100 and that the letter grades are A, B, C, and F.

LETTER GRADE	RANGE OF NUMERIC GRADES
A	All grades above 89
B	All grades above 79 and below 90
C	All grades above 69 and below 80
F	All grades below 70

[TABLE 3.3] A simple grading scheme

Expressed in English, an algorithm that uses this scheme would state that if the numeric grade is greater than 89, then the letter grade is A, else if the numeric grade is greater than 79, then the letter grade is B, ..., else (as a default case) the letter grade is F.

The process of testing several conditions and responding accordingly can be described in code by a **multi-way selection statement**. Here is a short Python script that uses such a statement to determine and print the letter grade corresponding to an input numeric grade:

```
number = int(input("Enter the numeric grade: "))
if number > 89:
    letter = 'A'
elif number > 79:
    letter = 'B'
elif number > 69:
    letter = 'C'
else:
    letter = 'F'
print("The letter grade is", letter)
```

The multi-way **if** statement considers each condition until one evaluates to **True** or they all evaluate to **False**. When a condition evaluates to **True**, the corresponding action is performed and control skips to the end of the entire selection statement. If no condition evaluates to **True**, then the action after the trailing **else** is performed.

The syntax of the multi-way **if** statement is the following:

```
if <condition-1>:
    <sequence of statements-1>

elif <condition-n>:
    <sequence of statements-n>
else:
    <default sequence of statements>
```

Once again, indentation helps the human reader and the Python interpreter to see the logical structure of this control statement.

3.4.5 Logical Operators and Compound Boolean Expressions

Often a course of action must be taken if either of two conditions is true. For example, valid inputs to a program often lie within a given range of values. Any input above this range should be rejected with an error message, and any input below this range should be dealt with in a similar fashion. The next code segment accepts only valid inputs for our grade conversion script and displays an error message otherwise:

```
number = int(input("Enter the numeric grade: "))
if number > 100:
    print("Error: grade must be between 100 and 0")
elif number < 0:
    print("Error: grade must be between 100 and 0")
else:
    # The code to compute and print the result goes here
```

Note that the first two conditions are associated with identical actions. Put another way, if either the first condition is true or the second condition is true, the program outputs the same error message. The two conditions can be combined in a Boolean expression that uses the **logical operator or**. The resulting **compound Boolean expression** simplifies the code somewhat, as follows:

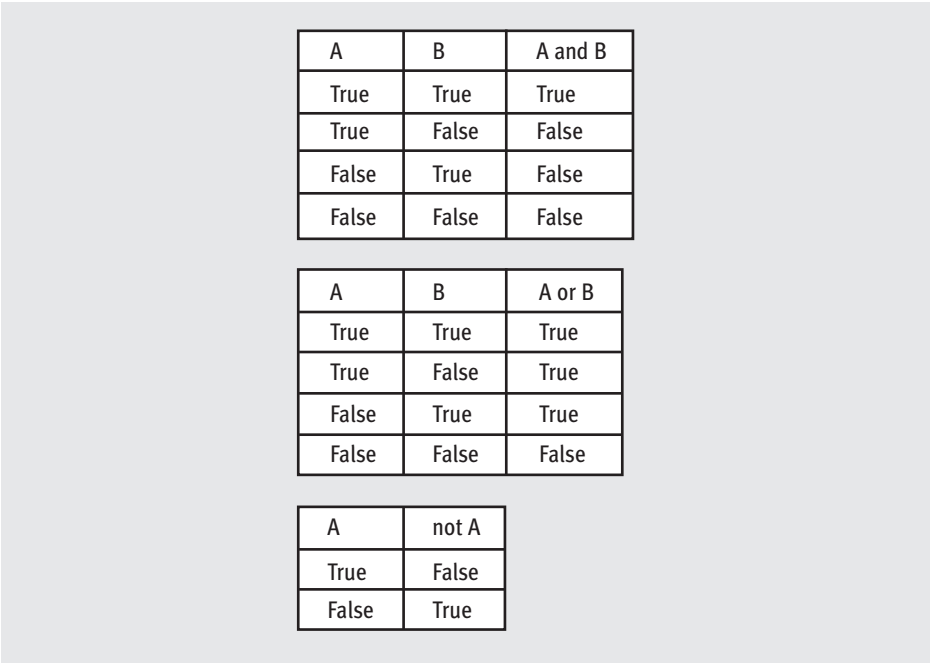
```
number = int(input("Enter the numeric grade: "))
if number > 100 or number < 0:
    print("Error: grade must be between 100 and 0")
else:
    # The code to compute and print the result goes here
```

Yet another way to describe this situation is to say that if the number is greater than or equal to 0 and less than or equal to 100, then we want the program to perform the computations and output the result; otherwise, it should output an error message. The logical operator **and** can be used to construct a different compound Boolean expression to express this logic:

```
number = int(input("Enter the numeric grade: "))
if number >= 0 and number <= 100:
    # The code to compute and print the result goes here
else:
    print("Error: grade must be between 100 and 0")
```

Python actually includes all three Boolean or logical operators, **and**, **or**, and **not**. Both the **and** operator and the **or** operator expect two operands. The **and** operator returns **True** if and only if both of its operands are true, and returns **False** otherwise. The **or** operator returns **False** if and only if both of its operands are false, and returns **True** otherwise. The **not** operator expects a single operand and returns its **logical negation**, **True**, if it's false, and **False** if it's true.

The behavior of each operator can be completely specified in a **truth table** for that operator. Each row below the first one in a truth table contains one possible combination of values for the operands and the value resulting from applying the operator to them. The first row contains labels for the operands and the expression being computed. Figure 3-4 shows the truth tables for **and**, **or**, and **not**.



[FIGURE 3.4] The truth tables for **and**, **or**, and **not**

The next example verifies some of the claims made in the truth tables in Figure 3-4:

```
>>> A = True
>>> B = False
>>> A and B
False
>>> A or B
True
>>> not A
False
```

The logical operators are evaluated after comparisons but before the assignment operator. The **not** operator has a higher precedence than both the **and** operator and the **or** operator, which have the same precedence. Thus, in our example, **not A and B** evaluates to **False**, whereas **not (A and B)** evaluates to **True**. Table 3-4 summarizes the precedence of the operators discussed thus far in this book.

TYPE OF OPERATOR	OPERATOR SYMBOL
Exponentiation	**
Arithmetic negation	-
Multiplication, division, remainder	*, /, %
Addition, subtraction	+, -
Comparison	==, !=, <, >, <=, >=
Logical negation	not
Logical conjunction and disjunction	and, or
Assignment	=

[TABLE 3-4] Operator precedence, from highest to lowest

3.4.6 Short-Circuit Evaluation

The Python virtual machine sometimes knows the value of a Boolean expression before it has evaluated all of its parts. For instance, in the expression **(A and B)**, if **A** is false, then so is the expression, and there is no need to evaluate **B**.

Likewise, in the expression (**A or B**), if **A** is true, then so is the expression, and again there is no need to evaluate **B**. This approach, in which evaluation stops as soon as possible, is called **short-circuit evaluation**.

There are times when short-circuit evaluation is advantageous. Consider the following example:

```
count = int(input("Enter the count: "))
sum = int(input("Enter the sum: "))
if count > 0 and sum // count > 10:
    print("average > 10")
else:
    print("count = 0 or average <= 10")
```

If the user enters 0 for the count, the condition contains a potential division by zero; however, because of short-circuit evaluation the division by zero is avoided.

3.4.7 Testing Selection Statements

Because selection statements add extra logic to a program, they open the door for extra logic errors. Thus, take special care when testing programs that contain selection statements.

The first rule of thumb is to make sure that all of the possible branches or alternatives in a selection statement are exercised. This will happen if the test data include values that make each condition true and also each condition false. In our grade-conversion example, the test data should definitely include numbers that produce each of the letter grades.

After testing all of the actions, you should also examine all of the conditions. For example, when a condition contains a single comparison of two numbers, try testing the program with operands that are equal, with a left operand that is less by one, and with a left operand that is greater by one, to catch errors in the boundary cases.

Finally, you need to test conditions that contain compound Boolean expressions using data that produce all of the possible combinations of values of the operands. As a blueprint for testing a compound Boolean expression, use the truth table for that expression.

3.4

Exercises

- 1 Assume that **x** is 3 and **y** is 5. Write the values of the following expressions:
 - a `x == y`
 - b `x > y - 3`
 - c `x <= y - 2`
 - d `x == y or x > 2`
 - e `x != 6 and y > 10`
 - f `x > 0 and x < 100`
- 2 Assume that **x** refers to a number. Write a code segment that prints the number's absolute value without using Python's **abs** function.
- 3 Write a loop that counts the number of space characters in a string. Recall that the space character is represented as ' '.
- 4 Assume that the variables **x** and **y** refer to strings. Write a code segment that prints these strings in alphabetical order. You should assume that they are not equal.
- 5 Explain how to check for an invalid input number and prevent it being used in a program. You may assume that the user enters a number.
- 6 Construct truth tables for the following Boolean expressions:
 - a `not (A or B)`
 - b `not A and not B`
- 7 Explain the role of the trailing **else** part of an extended **if** statement.
- 8 The variables **x** and **y** refer to numbers. Write a code segment that prompts the user for an arithmetic operator and prints the value obtained by applying that operator to **x** and **y**.
- 9 Does the Boolean expression `count > 0 and total // count > 0` contain a potential error? If not, why not?

3.5 Conditional Iteration: The `while` Loop

Earlier we examined the **for** loop, which executes a set of statements a definite number of times specified by the programmer. In many situations, however, the number of iterations in a loop is unpredictable. The loop eventually completes its work, but only when a condition changes. For example, the user might be asked for a set of input values. In that case, only the user knows the number she will enter. The program's input loop accepts these values until the user enters a special value or **sentinel** that terminates the input. This type of process is called **conditional iteration**. In this section, we explore the use of the **while** loop to describe conditional iteration.

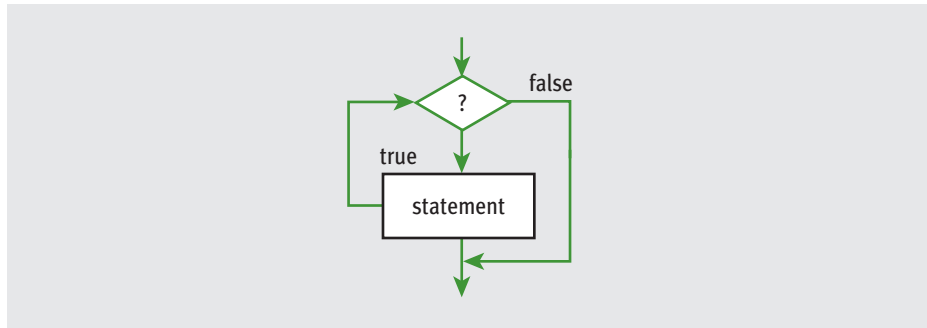
3.5.1 The Structure and Behavior of a `while` Loop

Conditional iteration requires that a condition be tested within the loop to determine whether the loop should continue. Such a condition is called the loop's **continuation condition**. If the continuation condition is false, the loop ends. If the continuation condition is true, the statements within the loop are executed again. The **while** loop is tailor-made for this type of control logic. Here is its syntax:

```
while <condition>:  
    <sequence of statements>
```

The form of this statement is almost identical to that of the one-way selection statement. However, the use of the reserved word **while** instead of **if** indicates that the sequence of statements might be executed many times, as long as the condition remains true.

Clearly, something eventually has to happen within the body of the loop to make the loop's continuation condition become false. Otherwise, the loop will continue forever, an error known as an **infinite loop**. At least one statement in the body of the loop must update a variable that affects the value of the condition. Figure 3-5 shows a flow diagram for the semantics of a **while** loop.



[FIGURE 3.5] The semantics of a **while** loop

The following example is a short script that prompts the user for a series of numbers, computes their sum, and outputs the result. Instead of forcing the user to enter a definite number of values, the program stops the input process when the user simply presses the return or enter key. The program recognizes this value as the empty string. We first present a rough draft in the form of a pseudocode algorithm:

```
set the sum to 0.0
input a string
while the string is not the empty string
    convert the string to a float
    add the float to the sum
    input a string
print the sum
```

Note that there are two input statements, one just before the loop header and one at the bottom of the loop body. The first input statement initializes a variable to a value that the loop condition can test. This variable is also called the **loop control variable**. The second input statement obtains all of the other input values, including one that will terminate the loop. Note also that the input must be received as a string, not a number, so the program can test for an empty string. If the string is not empty, we assume that it represents a number, and we convert it

to a **float**. Here is the Python code for this script, followed by a trace of a sample run:

```
sum = 0.0
data = input("Enter a number or just enter to quit: ")
while data != "":
    number = float(data)
    sum += number
    data = input("Enter a number or just enter to quit: ")
print("The sum is", sum)

Enter a number or just enter to quit: 3
Enter a number or just enter to quit: 4
Enter a number or just enter to quit: 5
Enter a number or just enter to quit:
The sum is 12.0
```

On this run, there are four inputs, including the empty string. Now, suppose we run the script again, and the user enters the empty string at the first prompt. The **while** loop's condition is immediately false, and its body does not execute at all! The sum prints as 0.0, which is just fine.

The **while** loop is also called an **entry-control loop**, because its condition is tested at the top of the loop. This implies that the statements within the loop can execute zero or more times.

3.5.2 Count Control with a **while** Loop

You can also use a **while** loop for a count-controlled loop. The next two code segments show the same summations with a **for** loop and a **while** loop, respectively.

```
sum = 0
for count in range(1, 100001):
    sum += count
print(sum)

sum = 0
count = 1
while count <= 100000:
    sum += count
    count += 1
print(sum)
```

Although both loops produce the same result, there is a tradeoff. The second code segment is noticeably more complex. It includes a Boolean expression and two extra statements that refer to the **count** variable. This loop control variable must be explicitly initialized before the loop header and incremented in the loop body. The **count** variable must also be examined in the explicit continuation condition. This extra manual labor for the programmer is not only time-consuming, but also potentially a source of new errors in loop logic.

By contrast, a **for** loop specifies the control information concisely in the header and automates its manipulation behind the scenes. However, we will soon see problems for which a **while** loop is the only solution. Therefore, you must master the logic of **while** loops and also be aware of the logic errors that they could produce.

The next example shows two versions of a script that counts down, from an upper bound of 10 to a lower bound of 1. It's up to you to decide which one is easier to understand and write correctly.

```
for count in range(10, 0, -1):
    print(count, end=" ")

count = 10
while count >= 1:
    print(count, end=" ")
    count -= 1
```

3.5.3 The while True Loop and the break Statement

Although the **while** loop can be complicated to write correctly, it is possible to simplify its structure and thus improve its readability. The first example script of this section, which contained two input statements, is a good candidate for such improvement. This loop's structure can be simplified if we receive the first input inside the loop, and break out of the loop if a test shows that the continuation condition is false. This implies postponing the actual test until the middle of the loop. Python includes a **break** statement that will allow us to make this change in the program. Here is the modified script:

```
sum = 0.0
while True:
    data = input("Enter a number or just enter to quit: ")
```

continued

```

if data == "":
    break
number = float(data)
sum += number
print("The sum is", sum)

```

The first thing to note is that the loop's entry condition is the Boolean value **True**. Some readers may become alarmed at this condition, which seems to imply that the loop will never exit. However, this condition is extremely easy to write and guarantees that the body of the loop will execute at least once. Within this body, the input datum is received. It is then tested for the loop's **termination condition** in a one-way selection statement. If the user wants to quit, the input will equal the empty string, and the **break** statement will cause an exit from the loop. Otherwise, control continues beyond the selection statement to the next two statements that process the input.

Our next example modifies the input section of the grade-conversion program to continue taking input numbers from the user until she enters an acceptable value. The logic of this loop is similar to that of the previous example.

```

while True:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        break
    else:
        print("Error: grade must be between 100 and 0")
print(number)  # Just echo the valid input

```

A trial run with just this segment shows the following interaction:

```

Enter the numeric grade: 101
Error: grade must be between 100 and 0
Enter the numeric grade: -1
Error: grade must be between 100 and 0
Enter the numeric grade: 45
45

```

Some computer scientists argue that a **while True** loop with a delayed exit violates the spirit of the **while** loop. However, in cases where the body of the loop must execute at least once, this technique simplifies the code and actually makes the program's logic clearer. If you are not persuaded by this reasoning and still want to test for the continuation and exit at the top of the loop, you can use a

Boolean variable to control the loop. Here is a version of the numeric input loop that uses a Boolean variable:

```
done = False
while not done:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        done = True
    else:
        print("Error: grade must be between 100 and 0")
print(number)    # Just echo the valid input
```

For an interesting discussion of this issue, see Eric Roberts's article, "Loop Exits and Structured Programming: Reopening the Debate", *ACM SIGCSE Bulletin*, Volume 27, Number 1, March 1995, pp. 268–272.

3.5.4 Random Numbers

The choices our algorithms have made thus far have been completely determined by given conditions that are either true or false. Many situations, such as games, include some randomness in the choices that are made. For example, we might toss a coin to see who kicks off in a football game. There is an equal probability of a coin landing heads-up or tails-up. Likewise, the roll of a die in many games entails an equal probability of the numbers 1 through 6 landing face-up. To simulate this type of randomness in computer applications, programming languages include resources for generating **random numbers**. Python's **random** module supports several ways to do this, but the easiest is to call the function **randint** with two integer arguments. The function **randint** returns a random number from among the numbers between the two arguments and including those numbers. The next session simulates the roll of a die 10 times:

```
>>> import random
>>> for roll in range(10):
    print(random.randint(1, 6), end=" ")

2 4 6 4 3 2 3 6 2 2
>>>
```

Although some values are repeated in this small set of calls, over the course of a large number of calls, the distribution of values approaches true randomness.

We can now use **randint**, selection, and a loop to develop a simple guessing game. At start-up, the user enters the smallest number and the largest number in the range. The computer then selects a number from this range. On each pass through the loop, the user enters a number in an attempt to guess the number selected by the computer. The program responds by saying “You’ve got it,” “Too large, try again,” or “Too small, try again.” When the user finally guesses the correct number, the program congratulates him and tells him the total number of guesses. Here is the code, followed by a sample run:

```
import random

smaller = int(input("Enter the smaller number: "))
larger = int(input("Enter the larger number: "))
myNumber = random.randint(smaller, larger)
count = 0
while True:
    count += 1
    userNumber = int(input("Enter your guess: "))
    if userNumber < myNumber:
        print("Too small")
    elif userNumber > myNumber:
        print("Too large")
    else:
        print("Congratulations! You've got it in", count, "tries!")
        break

Enter the smaller number: 1
Enter the larger number: 100
Enter your guess: 50
Too small
Enter your guess: 75
Too large
Enter your guess: 63
Too small
Enter your guess: 69
Too large
Enter your guess: 66
Too large
Enter your guess: 65
You've got it in 6 tries!
```


3.5.5 Loop Logic, Errors, and Testing

Because **while** loops can be the most complex control statements, to ensure their correct behavior, careful design and testing are needed. Testing a **while** loop must combine elements of testing used with **for** loops and with selection statements.

Errors to rule out during testing the **while** loop include an incorrectly initialized loop control variable, failure to update this variable correctly within the loop, and failure to test it correctly in the continuation condition. Moreover, if one simply forgets to update the control variable, the result is an infinite loop, which does not even qualify as an algorithm! To halt a loop that appears to be hung during testing, type **Control+C** in the terminal window or in the IDLE shell.

Genuine condition-controlled loops can be easy to design and test. If the continuation condition is already available for examination at loop entry, check it there and provide test data that produce 0, 1, and at least 5 iterations.

If the loop must run at least once, use a **while True** loop and delay the examination of the termination condition until it becomes available in the body of the loop. Ensure that something occurs in the loop to allow the condition to be checked and a **break** statement to be reached eventually.

3.5 Exercises

- 1 Translate the following **for** loops to equivalent **while** loops:
 - a

```
for count in range(100):  
    print(count)
```
 - b

```
for count in range(1, 101):  
    print(count)
```
 - c

```
for count in range(100, 0, -1):  
    print(count)
```
- 2 The factorial of an integer N is the product of all of the integers between 1 and N , inclusive. Write a **while** loop that computes the factorial of a given integer N .
- 3 The $\log_2$ of a given number N is given by M in the equation $N = 2^M$. The value of M is approximately equal to the number of times N can be evenly divided by 2 until it becomes 0. Write a loop that computes this approximation of the $\log_2$ of a given number N .

- 4 Describe the purpose of the **break** statement and the type of problem for which it is well suited.
- 5 What is the maximum number of guesses necessary to guess correctly a given number between the numbers N and M ?
- 6 What happens when the programmer forgets to update the loop control variable in a **while** loop?

3.6 Case Study: Approximating Square Roots

Users of pocket calculators or Python's **math** module do not have to think about how to compute square roots, but the people who built those calculators or wrote the code for that module certainly did. In this case study, we open the hood and see how this might be done.

3.6.1 Request

Write a program that computes square roots.

3.6.2 Analysis

The input to this program is a positive floating-point number or an integer. The output is a floating-point number representing the square root of the input number. For purposes of comparison, we also output Python's estimate of the square root using **math.sqrt**. Here is the proposed user interface:

```
Enter a positive number: 3
The program's estimate: 1.73205081001
Python's estimate:      1.73205080757
```

3.6.3 Design

In the seventeenth century, Sir Isaac Newton discovered an algorithm for approximating the square root of a positive number. Recall that the square root y of a positive number x is the number y such that $y^2 = x$. Newton discovered that if

one's initial estimate of y is z , then a better estimate of y can be obtained by taking the average of z together with x/z . The estimate can be transformed by this rule again and again, until a satisfactory estimate is reached.

A quick session with the Python interpreter shows this method of successive approximations in action. We let x be 25 and our initial estimate, z , be 1. We then use Newton's method to reset z to a better estimate and examine z to check it for closeness to the actual square root, 5. Here is a transcript of our interaction:

```
>>> x = 25
>>> y = 5                # The actual square root of x
>>> z = 1                # Our initial approximation
>>> z = (z + x / z) / 2   # Our first improvement
>>> z
13.0
>>> z = (z + x / z) / 2   # Our second improvement
>>> z
7.0
>>> z = (z + x / z) / 2   # Our third improvement - got it!
>>> z
5.0
>>>
```

After three transformations, the value of z is exactly equal to 5, the square root of 25. To include cases of numbers, such as 2 and 10, with irrational square roots, we can use an initial guess of 1.0 to produce floating-point results.

We now develop an algorithm to automate the process of successive transformations, because there might be many of them, and we don't want to write them all. Exactly how many of these operations are required depends on how close we want our final approximation to be to the actual square root. This closeness value, called the tolerance, can be compared to the difference between the value of x and the square of our estimate at any given time. While this difference is greater than the tolerance, the process continues; otherwise, it stops. The tolerance is typically a small value, such as 0.000001.

Our algorithm allows the user to input the number, uses a loop to apply Newton's method to compute the square root, and prints this value. Here is the pseudocode, followed by an explanation:

```
set x to the user's input value
set tolerance to 0.000001
set estimate to 1.0
while True
    set estimate to (estimate + x / estimate) / 2
```

```
    set difference to abs(x - estimate ** 2)
    if difference <= tolerance:
        break
output the estimate
```

Because our initial estimate is 1.0, the loop must compute at least one new estimate. Therefore, we use a **while True** loop. This loop transforms the estimate before determining whether it is close enough to the tolerance value to stop the process. The process should stop when the difference between the square of our estimate and the original number becomes less than or equal to the tolerance value. Note that this difference may be positive or negative, so we use the **abs** function to obtain its absolute value before examining it.

A more orthodox use of the **while** loop would compare the difference to the tolerance in the loop header. However, the difference must then be initialized before the loop to a large and rather meaningless value. The algorithm presented here captures the logic of the method of successive approximations more cleanly and simply.

3.6.4 Implementation (Coding)

The code for this program is straightforward.

```
"""
Program: newton.py
Author: Ken

Compute the square root of a number.

1. The input is a number.

2. The outputs are the program's estimate of the square root
   using Newton's method of successive approximations, and
   Python's own estimate using math.sqrt.
"""

import math

# Receive the input number from the user
x = float(input("Enter a positive number: "))

# Initialize the tolerance and estimate
tolerance = 0.000001
estimate = 1.0
```

continued

```

# Perform the successive approximations
while True:
    estimate = (estimate + x / estimate) / 2
    difference = abs(x - estimate ** 2)
    if difference <= tolerance:
        break

# Output the result
print("The program's estimate:", estimate)
print("Python's estimate:      ", math.sqrt(x))

```

3.6.5 Testing

The valid inputs to this program are positive integers and floating-point numbers. The display of Python's own most accurate estimate of the square root provides a benchmark for assessing the correctness of our own algorithm. We should at least provide a couple of perfect squares, such as 4 and 9, as well as numbers whose square roots are inexact, such as 2 and 3. A number between 1 and 0, such as .25, should also be included. Because the accuracy of our algorithm also depends on the size of the tolerance, we might alter this value during testing as well.

Summary

- Control statements determine the order in which other statements are executed in a program.
- Definite iteration is the process of executing a set of statements a fixed, predictable number of times. The **for** loop is an easy and convenient control statement for describing a definite iteration.
- The **for** loop consists of a header and a set of statements called the body. The header contains information that controls the number of times that the body executes.
- The **for** loop can count through a series of integers. Such a loop is called a count-controlled loop.

- During the execution of a count-controlled **for** loop, the statements in the loop's body can reference the current value of the count using the loop header's variable.
- Python's **range** function generates the sequence of numbers in a count-controlled **for** loop. This function can receive one, two, or three arguments. A single argument M specifies a sequence of numbers 0 through $M - 1$. Two arguments M and N specify a sequence of numbers M through $N - 1$. Three arguments M , N , and S specify a sequence of numbers M up through $N - 1$, stepping by S , when S is positive, or M down through $N + 1$, stepping by S , when S is negative.
- The **for** loop can traverse and visit the values in a sequence. Example sequences are a string of characters and a list of numbers.
- A format string and its operator **%** allow the programmer to format data using a field width and a precision.
- An off-by-one error occurs when a loop does not perform the intended number of iterations, there being one too many or one too few. This error can be caused by an incorrect lower bound or upper bound in a count-controlled loop.
- Boolean expressions contain the values **True** or **False**, variables bound to these values, comparisons using the relational operators, or other Boolean expressions using the logical operators. Boolean expressions evaluate to **True** or **False** and are used to form conditions in programs.
- The logical operators **and**, **or**, and **not** are used to construct compound Boolean expressions. The values of these expressions can be determined by constructing truth tables.
- Python uses short-circuit evaluation in compound Boolean expressions. The evaluation of the operands of **or** stops at the first true value, whereas the evaluation of the operands of **and** stops at the first false value.
- Selection statements are control statements that enable a program to make choices. A selection statement contains one or more conditions and the corresponding actions. Instead of moving straight ahead to the next action, the computer examines a condition. If the condition is true, the computer performs the corresponding action and then moves to the action following the selection statement. Otherwise, the computer moves to the next condition if there is one or to the action following the selection statement.

- A two-way selection statement, also called an **if-else** statement, has a single condition and two alternative courses of action. A one-way selection statement, also called an **if** statement, has a single condition and a single course of action. A multi-way selection statement, also called an extended **if** statement, has at least two conditions and three alternative courses of action.
- Conditional iteration is the process of executing a set of statements while a condition is true. The iteration stops when the condition becomes false. Because it cannot always be anticipated when this will occur, the number of iterations usually cannot be predicted.
- A **while** loop is used to describe conditional iteration. This loop consists of a header and a set of statements called the body. The header contains the loop's continuation condition. The body executes as long as the continuation condition is true.
- The **while** loop is an entry-control loop. This means that the continuation condition is tested at loop entry, and if it is false, the loop's body will not execute. Thus, the **while** loop can describe zero or more iterations.
- The **break** statement can be used to exit a **while** loop from its body. The **break** statement is usually used when the loop must perform at least one iteration. The loop header's condition in that case is the value **True**. The **break** statement is nested in an **if** statement that tests for a termination condition.
- Any **for** loop can be converted to an equivalent **while** loop. In a count-controlled **while** loop, the programmer must initialize and update a loop control variable.
- An infinite loop occurs when the loop's continuation condition never becomes false and no other exit points are provided. The primary cause of infinite loops is the programmer's failure to update a loop control variable properly.
- The function **random.randint** returns a random number in the range specified by its two arguments.

REVIEW QUESTIONS

- 1 How many times does a loop with the header **for count in range(10):** execute the statements in its body?
 - a 9 times
 - b 10 times
 - c 11 times
- 2 A **for** loop is convenient for
 - a making choices in a program
 - b running a set of statements a predictable number of times
 - c counting through a sequence of numbers
 - d describing conditional iteration
- 3 What is the output of the loop **for count in range(5): print(count)**?
 - a 1 2 3 4 5
 - b 1 2 3 4
 - c 0 1 2 3 4
- 4 When the function **range** receives two arguments, what does the second argument specify?
 - a the last value of a sequence of integers
 - b the last value of a sequence of integers plus 1
 - c the last value of a sequence of integers minus 1
- 5 Consider the following code segment:

```
x = 5
y = 4
if x > y:
    print(y)
else:
    print(x)
```

What value does this code segment print?

- a 4
- b 5

6 A Boolean expression using the **and** operator returns **True** when

- a both operands are true
- b one operand is true
- c neither operand is true

7 By default the **while** loop is an

- a entry-controlled loop
- b exit-controlled loop

8 Consider the following code segment:

```
count = 5
while count > 1:
    print(count, end=" ")
    count -= 1
```

What is the output produced by this code?

- a 1 2 3 4 5
- b 2 3 4 5
- c 5 4 3 2 1
- d 5 4 3 2

9 Consider the following code segment:

```
count = 1
while count <= 10:
    print(count, end=" ")
```

Which of the following describes the error in this code?

- a The loop is off by 1.
- b The loop control variable is not properly initialized.
- c The comparison points the wrong way.
- d The loop is infinite.

10 Consider the following code segment:

```
sum = 0.0
while True:
```

```
number = input("Enter a number: ")
if number == "":
    break
sum += float(number)
```

How many iterations does this loop perform?

- a none
- b at least one
- c zero or more
- d ten

PROJECTS

- 1 Write a program that accepts the lengths of three sides of a triangle as inputs. The program output should indicate whether or not the triangle is an equilateral triangle.
- 2 Write a program that accepts the lengths of three sides of a triangle as inputs. The program output should indicate whether or not the triangle is a right triangle. Recall from the Pythagorean theorem that in a right triangle, the square of one side equals the sum of the squares of the other two sides.
- 3 Modify the guessing-game program of Section 3.5 so that the user thinks of a number that the computer must guess. The computer must make no more than the minimum number of guesses.
- 4 A standard science experiment is to drop a ball and see how high it bounces. Once the “bounciness” of the ball has been determined, the ratio gives a bounciness index. For example, if a ball dropped from a height of 10 feet bounces 6 feet high, the index is 0.6, and the total distance traveled by the ball is 16 feet after one bounce. If the ball were to continue bouncing, the distance after two bounces would be $10\text{ ft} + 6\text{ ft} + 6\text{ ft} + 3.6\text{ ft} = 25.6\text{ ft}$. Note that the distance traveled for each successive bounce is the distance to the floor plus 0.6 of that distance as the ball comes back up. Write a program that lets the user enter the initial height of the ball and the number of times the ball is allowed to continue bouncing. Output should be the total distance traveled by the ball.

- 5 A local biologist needs a program to predict population growth. The inputs would be the initial number of organisms, the rate of growth (a real number greater than 0), the number of hours it takes to achieve this rate, and a number of hours during which the population grows. For example, one might start with a population of 500 organisms, a growth rate of 2, and a growth period to achieve this rate of 6 hours. Assuming that none of the organisms die, this would imply that this population would double in size every 6 hours. Thus, after allowing 6 hours for growth, we would have 1000 organisms, and after 12 hours, we would have 2000 organisms. Write a program that takes these inputs and displays a prediction of the total population.
- 6 The German mathematician Gottfried Leibniz developed the following method to approximate the value of π :
- $$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \dots$$
- Write a program that allows the user to specify the number of iterations used in this approximation and that displays the resulting value.
- 7 Teachers in most school districts are paid on a schedule that provides a salary based on their number of years of teaching experience. For example, a beginning teacher in the Lexington School District might be paid \$30,000 the first year. For each year of experience after this first year, up to 10 years, the teacher receives a 2% increase over the preceding value. Write a program that displays a salary schedule, in tabular format, for teachers in a school district. The inputs are the starting salary, the percentage increase, and the number of years in the schedule. Each row in the schedule should contain the year number and the salary for that year.
- 8 The greatest common divisor of two positive integers, A and B, is the largest number that can be evenly divided into both of them. Euclid's algorithm can be used to find the greatest common divisor (GCD) of two positive integers. You can use this algorithm in the following manner:
- Compute the remainder of dividing the larger number by the smaller number.
 - Replace the larger number with the smaller number and the smaller number with the remainder.
 - Repeat this process until the smaller number is zero.
- The larger number at this point is the GCD of A and B. Write a program that lets the user enter two integers and then prints each step in the process of using the Euclidean algorithm to find their GCD.

- 9 Write a program that receives a series of numbers from the user and allows the user to press the enter key to indicate that he or she is finished providing inputs. After the user presses the enter key, the program should print the sum of the numbers and their average.
- 10 The credit plan at TidBit Computer Store specifies a 10% down payment and an annual interest rate of 12%. Monthly payments are 5% of the listed purchase price, minus the down payment. Write a program that takes the purchase price as input. The program should display a table, with appropriate headers, of a payment schedule for the lifetime of the loan. Each row of the table should contain the following items:
- the month number (beginning with 1)
 - the current total balance owed
 - the interest owed for that month
 - the amount of principal owed for that month
 - the payment for that month
 - the balance remaining after payment

The amount of interest for a month is equal to $\text{balance} * \text{rate} / 12$. The amount of principal for a month is equal to the monthly payment minus the interest owed.

- 11 In the game of Lucky Sevens, the player rolls a pair of dice. If the dots add up to 7, the player wins \$4; otherwise, the player loses \$1. Suppose that, to entice the gullible, a casino tells players that there are lots of ways to win: (1, 6), (2, 5), etc. A little mathematical analysis reveals that there are not enough ways to win to make the game worthwhile; however, because many people's eyes glaze over at the first mention of mathematics, your challenge is to write a program that demonstrates the futility of playing the game. Your program should take as input the amount of money that the player wants to put into the pot, and play the game until the pot is empty. At that point, the program should print the number of rolls it took to break the player, as well as maximum amount of money in the pot.

[CHAPTER] 4 Strings and Text Files

After completing this chapter, you will be able to:

- Access individual characters in a string
- Retrieve a substring from a string
- Search for a substring in a string
- Convert a string representation of a number from one base to another base
- Use string methods to manipulate strings
- Open a text file for output and write strings or numbers to the file
- Open a text file for input and read strings or numbers from the file
- Use library functions to access and navigate a file system

Much about computation is concerned with manipulating text. Word processing and program editing are obvious examples, but text also forms the basis of e-mail, Web pages, and text messaging. In this chapter, we explore strings and text files, which are useful data structures for organizing and processing text.

4.1

Accessing Characters and Substrings in Strings

In Chapters 1 and 2 we used strings for input and output. We also combined strings via concatenation to form new strings. In Chapter 3, you learned how to format a string and to visit each of its characters with a **for** loop. In this section, we examine the internal structure of a string more closely, and you will learn how to extract portions of a string called **substrings**.

4.1.1

The Structure of Strings

Unlike an integer, which cannot be factored into more primitive parts, a string is a **data structure**. A data structure is a compound unit that consists of several smaller pieces of data. A string is a sequence of zero or more characters. When working with strings, the programmer sometimes must be aware of a string's length and the positions of the individual characters within the string. A string's length is the number of characters it contains. Python's **len** function returns this value when it is passed a string, as shown in the following session:

```
>>> len("Hi there!")
9
>>> len("")
0
>>>
```

The positions of a string's characters are numbered from 0, on the left, to the length of the string minus 1, on the right. Figure 4.1 illustrates the sequence of characters and their positions in the string **"Hi there!"**. Note that the ninth and last character, **!**, is at position 8.

H	i		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8

[FIGURE 4.1] Characters and their positions in a string

The string is an **immutable data structure**. This means that its internal data elements, the characters, can be accessed, but the structure itself cannot be modified.

4.1.2 The Subscript Operator

Although a simple **for** loop can access any of the characters in a string, sometimes you just want to inspect one character at a given position without visiting them all. The **subscript operator** makes this possible. The form of a subscript operator is the following:

```
<a string>[<an integer expression>]
```

The first part of the subscript operator is the string you want to inspect. The integer expression in brackets indicates the position of the particular character in the string that you want to inspect. The integer expression is also called an **index**. In the following examples, the subscript operator is used to access characters in the string “Alan Turing:”

```
>>> name = "Alan Turing"
>>> name[0]                                # Examine the first character
'A'
>>> name[3]                                # Examine the fourth character
'n'
>>> name[len(name)]                        # Oops! An index error!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> name[len(name) - 1]                    # Examine the last character
'g'
>>> name[-1]                              # Shorthand for the last one
'g'
>>>
```

Note that attempting to access a character using a position that equals the string's length results in an error. The positions usually range from 0 to the length minus 1. However, Python allows negative subscript values to access characters at or near the end of a string. The programmer counts backward from -1 to access characters from the right end of the string.

The subscript operator is also useful in loops where you want to use the positions as well as the characters in a string. The next code segment uses a count-controlled loop to display the characters and their positions:

```
>>> data = "Hi there!"
>>> for index in range(len(data)):
    print(index, data[index])

0 H
1 i
2
3 t
4 h
5 e
6 r
7 e
8 !
>>>
```

4.1.3 Slicing for Substrings

Some applications extract portions of strings called substrings. For example, an application that sorts filenames according to type might use the last three characters in a filename, called its **extension**, to determine the file's type (exceptions to this rule, such as the extensions **".py"** and **".html"**, will be considered later in this chapter). On a Windows file system, a filename ending in **".txt"** denotes a human-readable text file, whereas a filename ending in **".exe"** denotes an executable file of machine code. You can use Python's subscript operator to obtain a substring through a process called **slicing**. To extract a substring, the programmer places a colon (:) in the subscript. An integer value can appear on either side of the colon. Here are some examples that show how slicing is used:

```
>>> name = "myfile.txt"
>>> name[0:]           # The entire string
'myfile.txt'
>>> name[0:1]          # The first character
'm'
```

continued


```
>>> name[0:2]           # The first two characters
'my'
>>> name[:len(name)]   # The entire string
'myfile.txt'
>>> name[-3:]          # The last three characters
'txt'
>>>
```

Generally, when two integer positions are included in the slice, the range of characters in the substring extends from the first position up to but not including the second position. When the integer is omitted on either side of the colon, all of the characters extending to the end or the beginning are included in the substring. Note that the last line of code provides the correct range to obtain the filename's three-character extension.

4.1.4 Testing for a Substring with the `in` Operator

Another problem involves picking out strings that contain known substrings. For example, you might want to separate filenames with a `.txt` extension. A slice would work for this, but using Python's `in` operator is much simpler. When used with strings, the left operand of `in` is a target substring, and the right operand is the string to be searched. The operator `in` returns `True` if the target string is somewhere in the search string, or `False` otherwise. The next code segment traverses a list of filenames and prints just the filenames that have a `.txt` extension:

```
>>> fileList = ["myfile.txt", "myprogram.exe", "yourfile.txt"]
>>> for fileName in fileList:
    if ".txt" in fileName:
        print(fileName)

myfile.txt
yourfile.txt
>>>
```

4.1 Exercises

- 1 Assume that the variable **data** refers to the string **"myprogram.exe"**. Write the values of the following expressions:
 - a `data[2]`
 - b `data[-1]`
 - c `len(data)`
 - d `data[0:8]`
- 2 Assume that the variable **data** refers to the string **"myprogram.exe"**. Write the expressions that perform the following tasks:
 - a Extract the substring **"gram"** from **data**.
 - b Truncate the extension **".exe"** from **data**.
 - c Extract the character at the middle position from **data**.
- 3 Assume that the variable **myString** refers to a string. Write a code segment that uses a loop to print the characters of the string in reverse order.
- 4 Assume that the variable **myString** refers to a string, and the variable **reversedString** refers to an empty string. Write a loop that adds the characters from **myString** to **reversedString** in reverse order.

4.2 Data Encryption

As you might imagine, data traveling on the information highway is vulnerable to spies and potential thieves. It is easy to observe data crossing a network, particularly now that more and more communications involve wireless transmissions. For example, a person can sit in a car in the parking lot outside any major hotel and pick up transmissions between almost any two computers if that person runs the right **sniffing software**. For this reason, many applications now use **data encryption** to protect information transmitted on networks. Some application protocols have been updated to include secure versions that use data encryption. Examples of such versions are FTPS and HTTPS, which are secure versions of FTP and HTTP for file transfer and Web page transfer, respectively.

Encryption techniques are as old as the practice of sending and receiving messages. The sender **encrypts** a message by translating it to a secret code, called a **cipher text**. At the other end, the receiver **decrypts** the cipher text back to its original plaintext form. Both parties to this transaction must have at their disposal one or more **keys** that allow them to encrypt and decrypt messages. To give you a taste of this process, let us examine an encryption strategy in detail.

A very simple encryption method that has been in use for thousands of years is called a **Caesar cipher**. Recall that the character set for text is ordered as a sequence of distinct values. This encryption strategy replaces each character in the plaintext with the character that occurs a given distance away in the sequence. For positive distances, the method wraps around to the beginning of the sequence to locate the replacement characters for those characters near its end. For example, if the distance value of a Caesar cipher equals five characters, the string “invaders” would be encrypted as “nsafijwx.” To decrypt this cipher text back to plaintext, you apply a method that uses the same distance value but looks to the left of each character for its replacement. This decryption method wraps around to the end of the sequence to find a replacement character for one near its beginning.

The next two Python scripts implement Caesar cipher methods for any strings that contain lowercase letters and for any distance values between 0 and 26. Recall that the **ord** function returns the ordinal position of a character value in the ASCII sequence, whereas **chr** is the inverse function.

```
"""
File: encrypt.py
Encrypts an input string of lowercase letters and prints
the result. The other input is the distance value.
"""

plainText = input("Enter a one-word, lowercase message: ")
distance = int(input("Enter the distance value: "))
code = ""
for ch in plainText:
    ordValue = ord(ch)
    cipherValue = ordValue + distance
    if cipherValue > ord('z'):
        cipherValue = ord('a') + distance - \
            (ord('z') - ordValue + 1)
    code += chr(cipherValue)
print(code)
```

continued

```

"""
File: decrypt.py
Decrypts an input string of lowercase letters and prints
the result. The other input is the distance value.
"""

code = input("Enter the coded text: ")
distance = int(input("Enter the distance value: "))
plainText = ''
for ch in code:
    ordValue = ord(ch)
    cipherValue = ordValue - distance
    if cipherValue < ord('a'):
        cipherValue = ord('z') - \
            (distance - (ord('a') - ordValue + 1))
    plainText += chr(cipherValue)
print(plainText)

```

Here are some executions of the two scripts from a terminal command prompt. The user's inputs are in italics.

```

> python encrypt.py
Enter a one-word, lowercase message: invaders
Enter the distance value: 5
nsafijwx
> python decrypt.py
Enter the coded text: nsafijwx
Enter the distance value: 5
invaders

```

These scripts could easily be extended to cover all of the characters, including spaces and punctuation marks.

Although it worked reasonably well in ancient times, a Caesar cipher would be no match for a competent spy with a computer. Assuming that there are 128 ASCII characters, all you would have to do is write a program that would run the same line of text through the extended **decrypt** script with the values 0 through 127, until a meaningful plaintext is returned. It would take less than a second to do that on most modern computers. The main shortcoming of this encryption strategy is that the plaintext is encrypted one character at a time, and each encrypted character depends on that single character and a fixed distance value. In a sense, the structure of the original text is preserved in the cipher text, so it might not be hard to discover a key by visual inspection.

A more sophisticated encryption scheme is called a **block cipher**. A block cipher uses a plaintext character to compute two or more encrypted characters, and each encrypted character is computed using two or more plaintext characters. This is accomplished by using a mathematical structure known as an **invertible matrix** to determine the values of the encrypted characters. The matrix provides the key in this method. The receiver uses the same matrix to decrypt the cipher text. The fact that information used to determine each character comes from a block of data makes it more difficult to determine the key.

4.2 Exercises

- 1 Write the encrypted text of each of the following words using a Caesar cipher with a distance value of 3:
 - a python
 - b hacker
 - c wow
- 2 Consult the Table of ASCII values in Chapter 2 and suggest how you would modify the encryption and decryption scripts in this section to work with strings containing all of the printable characters.
- 3 You are given a string that was encoded by a Caesar cipher with an unknown distance value. The text can contain any of the printable ASCII characters. Suggest an algorithm for cracking this code.

4.3 Strings and Number Systems

When you perform arithmetic operations, you use the **decimal number system**. This system, also called the **base ten number system**, uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 as digits. As we saw in Chapter 1, the **binary number system** is used to represent all information in a digital computer. The two digits in this **base two number system** are 0 and 1. Because binary numbers can be long strings of 0s and 1s, computer scientists often use other number systems, such as **octal** (base eight) and **hexadecimal** (base 16) as shorthand for these numbers.

To identify the system being used, you attach the base as a subscript to the number. For example, the following numbers represent the quantity 415_{10} in the binary, octal, decimal, and hexadecimal systems:

415 in binary notation	110011111_2
415 in octal notation	637_8
415 in decimal notation	415_{10}
415 in hexadecimal notation	$19F_{16}$

The digits used in each system are counted from 0 to $n - 1$, where n is the system's base. Thus, the digits 8 and 9 do not appear in the octal system. To represent digits with values larger than 9_{10} , systems such as base 16 use letters. Thus, A_{16} represents the quantity 10_{10} , whereas 10_{16} represents the quantity 16_{10} . In this section, we examine how these systems actually represent numeric quantities and how to translate from one notation to another.

4.3.1 The Positional System for Representing Numbers

All of the number systems we have examined use **positional notation**—that is, the value of each digit in a number is determined by the digit's position in the number. In other words, each digit has a **positional value**. The positional value of a digit is determined by raising the base of the system to the power specified by the position ($base^{position}$). For an n -digit number, the positions (and exponents) are numbered from $n - 1$ down to 0, starting with the leftmost digit and moving to the right. For example, as Figure 4.2 illustrates, the positional values of the three-digit number 415_{10} are 100 (10^2), 10 (10^1), and 1 (10^0), moving from left to right in the number.

Positional values	100	10	1
Positions	2	1	0

[FIGURE 4.2] The first three positional values in the base 10 number system

To determine the quantity represented by a number in any system from base 2 through base 10, you multiply each digit (as a decimal number) by its positional

value and add the results. The following example shows how this is done for a three-digit number in base 10:

$$\begin{aligned} 415_{10} &= \\ 4 * 10^2 + 1 * 10^1 + 5 * 10^0 &= \\ 4 * 100 + 1 * 10 + 5 * 1 &= \\ 400 + 10 + 5 &= 415 \end{aligned}$$

4.3.2 Converting Binary to Decimal

Like the decimal system, the binary system also uses positional notation. However, each digit or bit in a binary number has a positional value that is a power of 2. In the discussion that follows, we occasionally refer to a binary number as a string of bits or a **bit string**. You determine the integer quantity that a string of bits represents in the usual manner: multiply the value of each bit (0 or 1) by its positional value and add the results. Let's do that for the number 1100111_2 :

$$\begin{aligned} 1100111_2 &= \\ 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 &= \\ 1 * 64 + 1 * 32 + 0 * 16 + 0 * 8 + 1 * 4 + 1 * 2 + 1 * 1 &= \\ 64 + 32 + 4 + 2 + 1 &= 103 \end{aligned}$$

Not only have we determined the integer value of this binary number, but we have also converted it to decimal in the process! In computing the value of a binary number, we can ignore the values of the positions occupied by 0s and simply add the positional values of the positions occupied by 1s.

We can code an algorithm for the conversion of a binary number to the equivalent decimal number as a Python script. The input to the script is a string of bits, and its output is the integer that the string represents. The algorithm uses a loop that accumulates the sum of a set of integers. The sum is initially 0. The exponent that corresponds to the position of the string's leftmost bit is the length of the bit string minus 1. The loop visits the digits in the string from the first to the last (left to right), also counting from the largest exponent of 2 down to 0 as

it goes. Each digit is converted to its integer value (1 or 0), multiplied by its positional value, and the result is added to the ongoing total. A positional value is computed by using the ****** operator. Here is the code for the script, followed by some example sessions at a terminal prompt:

```
"""
File: binarytodecimal.py
Converts a string of bits to a decimal integer.
"""

bstring = input("Enter a string of bits: ")
decimal = 0
exponent = len(bstring) - 1
for digit in bstring:
    decimal = decimal + int(digit) * 2 ** exponent
    exponent = exponent - 1
print("The integer value is", decimal)

> python binarytodecimal.py
Enter a string of bits: 1111
The integer value is 15
> python binarytodecimal.py
Enter a string of bits: 101
The integer value is 5
```

4.3.3 Converting Decimal to Binary

How are integers converted from decimal to binary? One algorithm uses division and subtraction instead of multiplication and addition. This algorithm repeatedly divides the decimal number by 2. After each division, the remainder (either a 0 or a 1) is placed at the beginning of a string of bits. The quotient becomes the next dividend in the process. The string of bits is initially empty, and the process continues while the decimal number is greater than 0.

Let's code this algorithm as a Python script and run it to display the intermediate results in the process. The script expects a non-negative decimal integer as an input and prints the equivalent bit string. The script checks first for a 0 and prints the string **'0'** as a special case. Otherwise, the script uses the algorithm

just described. On each pass through the loop, the values of the quotient, remainder, and result string are displayed. Here is the code for the script, followed by a session to convert the number 34:

```
"""
File: decimaltobinary.py
Converts a decimal integer to a string of bits.
"""

decimal = int(input("Enter a decimal integer: "))
if decimal == 0:
    print (0)
else:
    print("Quotient Remainder Binary")
    bstring = ""
    while decimal > 0:
        remainder = decimal % 2
        decimal = decimal // 2
        bstring = str(remainder) + bstring
        print("%5d%8d%12s" % (decimal, remainder, bstring))
    print("The binary representation is", bstring)

> python decimalToBinary.py
Enter a decimal integer: 34
Quotient Remainder Binary
17      0      0
8       1      10
4       0      010
2       0      0010
1       0      00010
0       1      100010
The binary representation is 100010
```

4.3.4 Conversion Shortcuts

There are various shortcuts for determining the decimal integer values of some binary numbers. One useful method involves learning to count through the numbers corresponding to the decimal values 0 through 8, as shown in Table 4.1.

DECIMAL	BINARY
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

[TABLE 4.1] The numbers 0 through 8 in binary

Note the rows that contain exact powers of 2 (2, 4, and 8 in decimal). Each of the corresponding binary numbers in that row contains a 1 followed by a number of zeroes that equal the exponent used to compute that power of 2. Thus, a quick way to compute the decimal value of the number 10000_2 is 2^4 or 16_{10} .

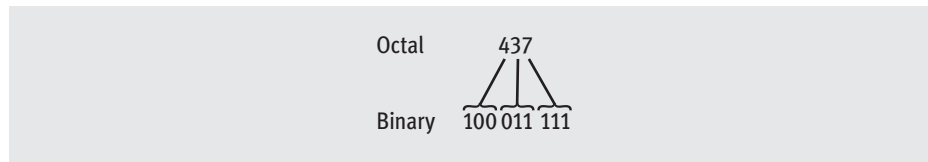
The rows whose binary numbers contain all ones correspond to decimal numbers that are one less than the next exact power of 2. For example, the number 111_2 equals $2^3 - 1$, or 7_{10} . Thus, a quick way to compute the decimal value of the number 11111_2 is $2^5 - 1$, or 31_{10} .

4.3.5

Octal and Hexadecimal Numbers

The octal system uses a base of eight and the digits 0...7. Conversions of octal to decimal and decimal to octal use algorithms similar to those discussed thus far (using powers of 8 and dividing by 8, instead of 2). But the real benefit of the octal system is the ease of converting octal numbers to and from binary. With practice, you can learn to do these conversions quite easily by hand, and in many cases by eye. To convert from octal to binary, you start by assuming that each digit in the octal number represents three digits in the corresponding binary number. You then start with the leftmost octal digit and write down the corresponding binary digits, padding these to the left with 0s to the count of 3, if

necessary. You proceed in this manner until you have converted all of the octal digits. Figure 4.3 shows such a conversion:

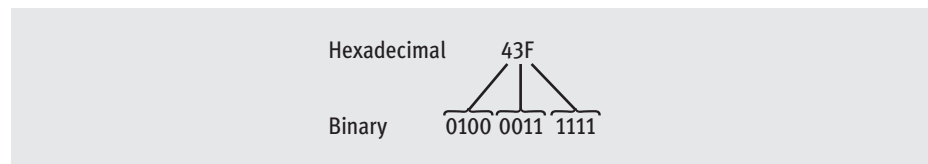


[FIGURE 4.3] The conversion of octal to binary

To convert binary to octal, you begin at the right and factor the bits into groups of three bits each. You then convert each group of three bits to the octal digit they represent.

As the size of a number system's base increases, so does the system's expressive power, its ability to say more with less. As bit strings get longer, the octal system becomes a less useful shorthand for expressing them. The hexadecimal or base-16 system (called "hex" for short), which uses 16 different digits, provides a more concise notation than octal for larger numbers. Base 16 uses the digits 0...9 for the corresponding integer quantities and the letters A...F for the integer quantities 10...15.

The conversion between numbers in the two systems works as follows. Each digit in the hexadecimal number is equivalent to four digits in the binary number. Thus, to convert from hexadecimal to binary, you replace each hexadecimal digit with the corresponding 4-bit binary number. To convert from binary to hexadecimal, you factor the bits into groups of four and look up the corresponding hex digits. (This is the kind of stuff that hackers memorize). Figure 4.4 shows a mapping of hexadecimal digits to binary digits.



[FIGURE 4.4] The conversion of hexadecimal to binary

4.3

Exercises

- 1 Translate each of the following numbers to decimal numbers:
 - a 11001_2
 - b 100000_2
 - c 11111_2
- 2 Translate each of the following numbers to binary numbers:
 - a 47_{10}
 - b 127_{10}
 - c 64_{10}
- 3 Translate each of the following numbers to binary numbers:
 - a 47_8
 - b 127_8
 - c 64_8
- 4 Translate each of the following numbers to decimal numbers:
 - a 47_8
 - b 127_8
 - c 64_8
- 5 Translate each of the following numbers to decimal numbers:
 - a 47_{16}
 - b 127_{16}
 - c AA_{16}

4.4

String Methods

Text processing involves many different operations on strings. For example, consider the problem of analyzing someone's writing style. Short sentences containing short words are generally considered more readable than long sentences containing long words. A program to compute a text's average sentence length and the average word length might provide a rough analysis of style.

Let's start with counting the words in a single sentence and finding the average word length. This task requires locating the words in a string. Fortunately, Python includes a set of string operations called **methods** that make tasks like this one easy. In the next session, we use the string method **split** to obtain a list of the words contained in an input string. We then print the length of the list, which equals the number of words, and compute and print the average of the lengths of the words in the list.

```
>>> sentence = input("Enter a sentence: ")
Enter a sentence: This sentence has no long words.
>>> listOfWords = sentence.split()
>>> print("There are", len(listOfWords), "words.")
There are 6 words.
>>> sum = 0
>>> for word in listOfWords:
    sum += len(word)

>>> print("The average word length is", sum / len(listOfWords))
The average word length is 4.5
>>>
```

A method behaves like a function, but has a slightly different syntax. Unlike a function, a method is always called with a given data value called an **object**, which is placed before the method name in the call. The syntax of a method call is the following:

```
<an object>.<method name>(<argument-1>, ..., <argument-n>)
```

Methods can also expect arguments and return values. A method knows about the internal state of the object with which it is called. Thus, the method **split** in our example builds a list of the words in the string object to which **sentence** refers and returns it.

In short, methods are as useful as functions, but you need to get used to the dot notation, which you have already seen when using a function associated with a module. In Python, all data values are in fact objects, and every data type includes a set of methods to use with objects of that type.

Table 4.2 lists some useful string methods. You can view the complete list and the documentation of the string methods by entering **dir(str)** or **help(str)** at a shell prompt. Note that some arguments are enclosed in square brackets (**[]**). These indicate that the arguments are optional and may be omitted when the method is called.

STRING METHOD	WHAT IT DOES
<code>s.center(width)</code>	Returns a copy of <code>s</code> centered within the given number of columns.
<code>s.count(sub [, start [, end]])</code>	Returns the number of non-overlapping occurrences of substring <code>sub</code> in <code>s</code> . Optional arguments <code>start</code> and <code>end</code> are interpreted as in slice notation.
<code>s.endswith(sub)</code>	Returns <code>True</code> if <code>s</code> ends with <code>sub</code> or <code>False</code> otherwise.
<code>s.find(sub [, start [, end]])</code>	Returns the lowest index in <code>s</code> where substring <code>sub</code> is found. Optional arguments <code>start</code> and <code>end</code> are interpreted as in slice notation.
<code>s.isalpha()</code>	Returns <code>True</code> if <code>s</code> contains only letters or <code>False</code> otherwise.
<code>s.isdigit()</code>	Returns <code>True</code> if <code>s</code> contains only digits or <code>False</code> otherwise.
<code>s.join(sequence)</code>	Returns a string that is the concatenation of the strings in the sequence. The separator between elements is <code>s</code> .
<code>s.lower()</code>	Returns a copy of <code>s</code> converted to lowercase.
<code>s.replace(old, new [, count])</code>	Returns a copy of <code>s</code> with all occurrences of substring <code>old</code> replaced by <code>new</code> . If the optional argument <code>count</code> is given, only the first <code>count</code> occurrences are replaced.
<code>s.split([sep])</code>	Returns a list of the words in <code>s</code> , using <code>sep</code> as the delimiter string. If <code>sep</code> is not specified, any whitespace string is a separator.
<code>s.startswith(sub)</code>	Returns <code>True</code> if <code>s</code> starts with <code>sub</code> or <code>False</code> otherwise.
<code>s.strip([aString])</code>	Returns a copy of <code>s</code> with leading and trailing whitespace (tabs, spaces, newlines) removed. If <code>aString</code> is given, remove characters in <code>aString</code> instead.
<code>s.upper()</code>	Returns a copy of <code>s</code> converted to uppercase.

[TABLE 4.2] Some useful string methods, with the code letter **s** used to refer to any string

The next session shows these methods in action:

```
>>> s = "Hi there!"
>>> len(s)
9
>>> s.center(11)
' Hi there! '
>>> s.count('e')
2
>>> s.endswith("there!")
True
>>> s.startswith("Hi")
True
>>> s.find('the')
3
>>> s.isalpha()
False
>>> 'abc'.isalpha()
True
>>> "326".isdigit()
True
>>> words = s.split()
>>> words
['Hi', 'there!']
>>> "".join(words)
'Hithere!'
>>> " ".join(words)
'Hi there!'
>>> s.lower()
'hi there!'
>>> s.upper()
'HI THERE!'
>>> s.replace('i', 'o')
'Ho there!'
>>> " Hi there! ".strip()
'Hi there!'
>>>
```

Now that you know about the string method **split**, you are in a position to use a more general strategy for extracting a filename's extension than the one used earlier in this chapter. The method **split** returns a list of words in the string upon which it is called. This method assumes that the default separator

character between the words is a space. You can override this assumption by passing a period as an argument to **split**, as shown in the next session:

```
>>> "myfile.txt".split(".")
['myfile', 'txt']
>>> "myfile.py".split(".")
['myfile', 'py']
>>> "myfile.html".split(".")
['myfile', 'html']
>>>
```

Note that the extension, regardless of its length, is the last string in each list. You can now use the subscript **[-1]**, which also extracts the last element in a list, to write a general expression for obtaining any filename's extension, as follows:

```
filename.split(".")[-1]
```

4.4 Exercises

- 1 Assume that the variable **data** refers to the string **"Python rules!"**. Use a string method from Table 4.2 to perform the following tasks:
 - a Obtain a list of the words in the string.
 - b Convert the string to uppercase.
 - c Locate the position of the string **"rules"**.
 - d Replace the exclamation point with a question mark.
- 2 Using the value of **data** from Exercise 1, write the values of the following expressions:
 - a **data.endswith('i')**
 - b **" totally ".join(data.split())**

4.5 Text Files

Thus far in this book, we have seen examples of programs that have taken input data from users at the keyboard. Most of these programs can receive their input from text files as well. A text file is a software object that stores data on a permanent medium such as a disk, CD, or flash memory. When compared to keyboard input from a human user, the main advantages of taking input data from a file are the following:

- The data set can be much larger.
- The data can be input much more quickly and with less chance of error.
- The data can be used repeatedly with the same program or with different programs.

4.5.1 Text Files and Their Format

Using a text editor such as Notepad or TextEdit, you can create, view, and save data in a text file. Your Python programs can output data to a text file, a procedure explained later in this section. The data in a text file can be viewed as characters, words, numbers, or lines of text, depending on the text file's format and on the purposes for which the data are used. When the data are treated as numbers (either integers or floating-points), they must be separated by white-space characters—spaces, tabs, and newlines. For example, a text file containing six floating-point numbers might look like

```
34.6 22.33 66.75  
77.12 21.44 99.01
```

when examined with a text editor. Note that this format includes a space or a newline as a separator of items in the text.

All data output to or input from a text file must be strings. Thus, numbers must be converted to strings before output, and these strings must be converted back to numbers after input.

4.5.2 Writing Text to a File

Data can be output to a text file using a **file** object. Python's **open** function, which expects a file pathname and a **mode string** as arguments, opens a connection to the file on disk and returns a **file** object. The mode string is **'r'** for input files and **'w'** for output files. Thus, the following code opens a **file** object on a file named **myfile.txt** for output:

```
>>> f = open("myfile.txt", 'w')
```

If the file does not exist, it is created with the given pathname. If the file already exists, Python opens it. When data are written to the file and the file is closed, any data previously existing in the file are erased.

String data are written (or output) to a file using the method **write** with the **file** object. The **write** method expects a single string argument. If you want the output text to end with a newline, you must include the escape character **\n** in the string. The next statement writes two lines of text to the file:

```
>>> f.write("First line.\nSecond line.\n")
```

When all of the outputs are finished, the file should be closed using the method **close**, as follows:

```
>>> f.close()
```

Failure to close an output file can result in data being lost.

4.5.3 Writing Numbers to a File

The **file** method **write** expects a string as an argument. Therefore, other types of data, such as integers or floating-point numbers, must first be converted to strings before being written to an output file. In Python, the values of most data types can be converted to strings by using the **str** function. The resulting strings are then written to a file with a space or a newline as a separator character.

The next code segment illustrates the output of integers to a text file. Five hundred random integers between 1 and 500 are generated and written to a text file named **integers.txt**. The newline character is the separator.

```
import random
f = open("integers.txt", 'w')
for count in range(500):
    number = random.randint(1, 500)
    f.write(str(number) + "\n")
f.close()
```

4.5.4 Reading Text from a File

You open a file for input in a manner similar to opening a file for output. The only thing that changes is the mode string, which, in the case of opening a file for input, is **'r'**. However, if the pathname is not accessible from the current working directory, Python raises an error. Here is the code for opening **myfile.txt** for input:

```
>>> f = open("myfile.txt", 'r')
```

There are several ways to read data from an input file. The simplest way is to use the **file** method **read** to input the entire contents of the file as a single string. If the file contains multiple lines of text, the newline characters will be embedded in this string. The next session shows how to use the method **read**:

```
>>> text = f.read()
>>> text
'First line.\nSecond line.\n'
>>> print(text)
First line.
Second line.

>>>
```

After input is finished, another call to **read** would return an empty string, to indicate that the end of the file has been reached. To repeat an input, the file must be re-opened. It is not necessary to close the file.

Alternatively, an application might read and process the text one line at a time. A **for** loop accomplishes this nicely. The **for** loop views a **file** object as a sequence of lines of text. On each pass through the loop, the loop variable is bound to the next line of text in the sequence. Here is a session that re-opens our example file and visits the lines of text in it:

```
>>> f = open("myfile.txt", 'r')
>>> for line in f:
    print(line)

First line.

Second line.

>>>
```

Note that **print** appears to output an extra newline. This is because each line of text input from the file retains its newline character.

In cases where you might want to read a specified number of lines from a file (say, the first line only), you can use the **file** method **readline**. The **readline** method consumes a line of input and returns this string, including the newline. If **readline** encounters the end of the file, it returns the empty string. The next code segment uses our old friend the **while True** loop to input all of the lines of text with **readline**:

```
>>> f = open("myfile.txt", 'r')
>>> while True:
    line = f.readline()
    if line == "":
        break
    print(line)

First line.

Second line.

>>>
```

4.5.5 Reading Numbers from a File

All of the **file** input operations return data to the program as strings. If these strings represent other types of data, such as integers or floating-point numbers, the programmer must convert them to the appropriate types before manipulating them further. In Python, the string representations of integers and floating-point numbers can be converted to the numbers themselves by using the functions **int** and **float**, respectively.

When reading data from a file, another important consideration is the format of the data items in the file. Earlier, we showed an example code segment that output integers separated by newlines to a text file. During input, these data can be read with a simple **for** loop. This loop accesses a line of text on each pass. To convert this line to the integer contained in it, the programmer runs the string method **strip** to remove the newline and then runs the **int** function to obtain the integer value.

The next code segment illustrates this technique. It opens the file of random integers written earlier, reads them, and prints their sum.

```
f = open("integers.txt", 'r')
sum = 0
for line in f:
    line = line.strip()
    number = int(line)
    sum += number
print("The sum is", sum)
```

Obtaining numbers from a text file in which they are separated by spaces is a bit trickier. One method proceeds by reading lines in a **for** loop, as before. But each line now can contain several integers separated by spaces. You can use the string method **split** to obtain a list of the strings representing these integers, and then process each string in this list with another **for** loop.

The next code segment modifies the previous one to handle integers separated by spaces and/or newlines.

```
f = open("integers.txt", 'r')
sum = 0
for line in f:
    wordlist = line.split()
    for word in wordlist:
        number = int(word)
        sum += number
print("The sum is", sum)
```

Note that the line does not have to be stripped of the newline, because **split** takes care of that automatically.

Table 4.3 summarizes the **file** operations discussed in this section. Note that the dot notation is not used with **open**, which returns a new **file** object.

METHOD	WHAT IT DOES
open(pathname, mode)	Opens a file at the given pathname and returns a file object. The mode can be 'r' , 'w' , 'rw' , or 'a' . The last two values, 'rw' and 'a' , mean read/write and append, respectively.
f.close()	Closes an output file. Not needed for input files.
f.write(aString)	Outputs aString to a file.
f.read()	Inputs the contents of a file and returns them as a single string. Returns '' if the end of file is reached.
f.readline()	Inputs a line of text and returns it as a string, including the newline. Returns '' if the end of file is reached.

[TABLE 4.3] Some **file** operations

4.5.6 Accessing and Manipulating Files and Directories on Disk

When designing Python programs that interact with files, it's a good idea to include error recovery. For example, before attempting to open a file for input, the programmer should check to see if a file with the given pathname exists on the disk. Tables 4.4 and 4.5 explain some file system functions, including a function (**os.path.exists**) that supports this checking. They also list some functions that allow your programs to navigate to a given directory in the file system, as well as perform some disk housekeeping. The functions listed in Tables 4.4 and 4.5 are self-explanatory, and you are encouraged to experiment. For example, the

following code segment will print all of the names of files in the current working directory that have a **.py** extension:

```
import os
currentDirectoryPath = os.getcwd()
listOfFileNames = os.listdir(currentDirectoryPath)
for name in listOfFileNames:
    if ".py" in name:
        print(name)
```

os MODULE FUNCTION	WHAT IT DOES
chdir(path)	Changes the current working directory to path .
getcwd()	Returns the path of the current working directory.
listdir(path)	Returns a list of the names in directory named path .
mkdir(path)	Creates a new directory named path and places it in the current working directory.
remove(path)	Removes the file named path from the current working directory.
rename(old, new)	Renames the file or directory named old to new .
rmdir(path)	Removes the directory named path from the current working directory.

[TABLE 4.4] Some file system functions

os.path MODULE FUNCTION	WHAT IT DOES
exists(path)	Returns True if path exists and False otherwise.
isdir(path)	Returns True if path names a directory and False otherwise.
isfile(path)	Returns True if path names a file and False otherwise.
getsize(path)	Returns the size of the object names by path in bytes.

[TABLE 4.5] More file system functions

4.5 Exercises

- 1 Write a code segment that opens a file named **myfile.txt** for input and prints the number of lines in the file.
- 2 Write a code segment that opens a file for input and prints the number of four-letter words in the file.
- 3 Assume that a file contains integers separated by newlines. Write a code segment that opens the file and prints the average value of the integers.
- 4 Write a code segment that prints the names of all of the items in the current working directory.
- 5 Write a code segment that prompts the user for a filename. If the file exists, the program should print its contents on the terminal. Otherwise, it should print an error message.

4.6 Case Study: Text Analysis

In 1949, Dr. Rudolf Flesch published *The Art of Readable Writing*, in which he proposed a measure of text readability known as the **Flesch Index**. This index is based on the average number of syllables per word and the average number of words per sentence in a piece of text. Index scores usually range from 0 to 100, and indicate readable prose for the following grade levels:

FLESCH INDEX	GRADE LEVEL OF READABILITY
0–30	College
50–60	High School
90–100	Fourth Grade

In this case study, we develop a program that computes the Flesch Index for a text file.

4.6.1 Request

Write a program that computes the Flesch Index and grade level for text stored in a text file.

4.6.2 Analysis

The input to this program is the name of a text file. The outputs are the number of sentences, words, and syllables in the file, as well as the file’s Flesch Index and Grade Level Equivalent.

During analysis, we consult experts in the problem domain to learn any information that might be relevant in solving the problem. For our problem, this information includes the definitions of *sentence*, *word*, and *syllable*. For the purposes of this program, these terms are defined in Table 4.6.

Word	Any sequence of non-whitespace characters.
Sentence	Any sequence of words ending in a period, question mark, exclamation point, colon, or semicolon.
Syllable	Any word of three characters or less; or any vowel (a, e, i, o, u) or pair of consecutive vowels, except for a final -es, -ed, or -e that is not -le.

[TABLE 4.6] Definitions of items used in the text-analysis program

Note that the definitions of *word* and *sentence* are approximations. Some words, such as “doubles” and “syllables,” end in “es” but will be counted as having one syllable, and an ellipse (“...”) will be counted as three sentences.

Flesch’s formula to calculate the index *F* is the following:

$$F = 206.835 - 1.015 \times (\text{words} / \text{sentences}) - 84.6 \times (\text{syllables} / \text{words})$$

The **Flesch-Kincaid Grade Level Formula** is used to compute the Equivalent Grade Level *G*:

$$G = 0.39 \times (\text{words} / \text{sentences}) + 11.8 \times (\text{syllables} / \text{words}) - 15.59$$

4.6.3 Design

This program will perform the following tasks:

- 1 Receive the filename from the user, open the file for input, and input the text.
- 2 Count the sentences in the text.
- 3 Count the words in the text.
- 4 Count the syllables in the text.
- 5 Compute the Flesch Index.
- 6 Compute the Grade Level Equivalent.
- 7 Print these two values with the appropriate labels, as well as the counts from tasks 2–4.

The first and last tasks require no design. Let's assume that the text is input as a single string from the file and is then processed in tasks 2–4. These three tasks can be designed as code segments that use the input string and produce an integer value. Task 5, computing the Flesch Index, uses the three integer results of tasks 2–4 to compute the Flesch Index. Lastly, task 6 is a code segment that uses the same integers and computes the Grade Level Equivalent. The five tasks are listed in Table 4.7, where **text** is a variable that refers to the string read from the file.

TASK	WHAT IT DOES
count the sentences	Counts the number of sentences in text .
count the words	Counts the number of words in text .
count the syllables	Counts the number of syllables in text .
compute the Flesch Index	Computes the Flesch Index for the given numbers of sentences, words, and syllables.
compute the grade level	Computes the Grade Level Equivalent for the given numbers of sentences, words, and syllables.

[TABLE 4.7] The tasks defined in the text analysis program

All the real work is done in the tasks that count the items:

- Add the number of characters in **text** that end the sentences. These characters were specified in analysis, and the string method **count** is used to count them in the algorithm.
- Split **text** into a list of words and determine the **text** length.
- Count the syllables in each word in **text**.

The last task is the most complex. For each word in the text, we must count the syllables in that word. From analysis, we know that each distinct vowel counts as a syllable, unless it is in the endings -ed, -es, or -e (but not -le). For now, we ignore the possibility of consecutive vowels.

4.6.4 Implementation (Coding)

The main tasks are marked off in the program code with a blank line and a comment.

```
"""
Program: textanalysis.py
Author: Ken
Computes and displays the Flesch Index and the Grade
Level Equivalent for the readability of a text file.
"""

# Take the inputs
fileName = input("Enter the file name: ")
inputFile = open(fileName, 'r')
text = inputFile.read()

# Count the sentences
sentences = text.count('.') + text.count('?') + \
            text.count(':') + text.count(';') + \
            text.count('!')

# Count the words
words = len(text.split())

# Count the syllables
syllables = 0
for word in text.split():
    for vowel in ['a', 'e', 'i', 'o', 'u']:
        syllables += word.count(vowel)
```

continued

```

for ending in ['es', 'ed', 'e']:
    if word.endswith(ending):
        syllables -= 1
if word.endswith('le'):
    syllables += 1

# Compute the Flesch Index and Grade Level
index = 206.835 - 1.015 * (words / sentences) - \
        84.6 * (syllables / words)
level = round(0.39 * (words / sentences) + 11.8 * \
              (syllables / words) - 15.59)

# Output the results
print("The Flesch Index is", index)
print("The Grade Level Equivalent is", level)
print(sentences, "sentences")
print(words, "words")
print(syllables, "syllables")

```

4.6.5 Testing

Although the main tasks all collaborate in the text analysis program, they can be tested more or less independently, before the entire program is tested. After all, there is no point in running the complete program if you are unsure that even one of the tasks does not work correctly.

This kind of procedure is called **bottom-up testing**. Each task is coded and tested before it is integrated into the overall program. After you have written code for one or two tasks, you can test them in a short script. This script is called a **driver**. For example, here is a driver that tests the code for computing the Flesch Index and the Grade Level Equivalent without using a text file:

```

"""
Program: fleschdriver.py
Author: Ken
Test driver for Flesch Index and Grade level.
"""

sentences = int(input("Sentences: "))
words = int(input("Words: "))
syllables = int(input("Syllables: "))

```

continued

```

index = 206.835 - 1.015 * (words / sentences) - \
        84.6 * (syllables / words)
print("Flesch Index:", index)
level = round(0.39 * (words / sentences) + 11.8 * \
              (syllables / words) - 15.59)
print("Grade Level: ", level)

```

This driver allows the programmer not only to verify the two tasks, but also to obtain some data to use when testing the complete program later on. For example, the programmer can supply a text file that contains the number of sentences, words, and syllables already tested in the driver, and then compare the two test results.

In bottom-up testing, the lower-level tasks must be developed and tested before those tasks that depend on the lower-level tasks.

When you have tested all of the parts you can integrate them into the complete program. The test data at that point should be short files that produce the expected results. Then, you should use longer files. For example, you might see if plaintext versions of Dr. Seuss's *Green Eggs and Ham* and Shakespeare's *Hamlet* produce grade levels of 5th grade and 12th grade, respectively. Or you could test the program with its own source program file—but we predict that its readability will seem quite low, because it lacks most of the standard end-of-sentence marks!

Summary

- A string is a sequence of zero or more characters. The **len** function returns the number of characters in its string argument. Each character occupies a position in the string. The positions range from 0 to the length of the string minus 1.
- A string is an immutable data structure. Its contents can be accessed, but its structure cannot be modified.
- The subscript operator **[]** can be used to access a character at a given position in a string. The operand or index inside the subscript operator must be an integer expression whose value is less than the string's length. A negative index can be used to access a character at or near the end of the string, starting with -1.

- A subscript operator can also be used for slicing—to fetch a substring from a string. When the subscript has the form [**<start>**:], the substring contains the characters from the **start** position to the end of the string. When the form is [:**<end>**], the positions range from the first one to **end - 1**. When the form is [**<start>**:**<end>**], the positions range from **start** to **end - 1**.
- The **in** operator is used to detect the presence or absence of a substring in a string. Its usage is **<substring> in <a string>**.
- A method is an operation that is used with an object. A method can expect arguments and return a value.
- The string type includes many useful methods for use with string objects.
- A text file is a software object that allows a program to transfer data to and from permanent storage on disk, CDs, or flash memory.
- A **file** object is used to open a connection to a text file for input or output.
- The **file** method **write** is used to output a string to a text file.
- The **file** method **read** inputs the entire contents of a text file as a single string.
- The **file** method **readline** inputs a line of text from a text file as a string.
- The **for** loop treats an input file as a sequence of lines. On each pass through the loop, the loop's variable is bound to a line of text read from the file.

REVIEW QUESTIONS

For questions 1–6, assume that the variable **data** refers to the string "**No way!**".

- 1 The expression **len(data)** evaluates to

a	8
b	7
c	6

- 2 The expression `data[1]` evaluates to
- a `'N'`
 - b `'o'`
- 3 The expression `data[-1]` evaluates to
- a `'!'`
 - b `'y'`
- 4 The expression `data[3:6]` evaluates to
- a `'way!'`
 - b `'way'`
 - c `' wa'`
- 5 The expression `data.replace("No", "Yes")` evaluates to
- a `'No way!'`
 - b `'Yo way!'`
 - c `'Yes way!'`
- 6 The expression `data.find("way!")` evaluates to
- a `2`
 - b `3`
 - c `True`
- 7 A Caesar cipher locates the coded text of a plaintext character
- a A given distance to the left or the right in the sequence of characters
 - b In an inversion matrix
- 8 The binary number 111 represents the decimal integer
- a `111`
 - b `3`
 - c `7`

- 9 Which of the following binary numbers represents the decimal integer value 8?
- a 11111111
 - b 100
 - c 1000
- 10 Which **file** method is used to read the entire contents of a file in a single operation?
- a **readline**
 - b **read**
 - c a **for** loop

PROJECTS

- 1 Write a script that inputs a line of plaintext and a distance value and outputs an encrypted text using a Caesar cipher. The script should work for any printable characters.
- 2 Write a script that inputs a line of encrypted text and a distance value and outputs plaintext using a Caesar cipher. The script should work for any printable characters.
- 3 Modify the scripts of Projects 1 and 2 to encrypt and decrypt entire files of text.
- 4 Octal numbers have a base of eight and the digits 0–7. Write the scripts **octalToDecimal.py** and **decimalToOctal.py**, which convert numbers between the octal and decimal representations of integers. These scripts use algorithms similar to those of the **binaryToDecimal** and **decimalToBinary** scripts developed in Section 4.3.
- 5 A **bit shift** is a procedure whereby the bits in a bit string are moved to the left or to the right. For example, we can shift the bits in the string **1011** two places to the left to produce the string **1110**. Note that the leftmost two bits are wrapped around to the right side of the string in this operation. Define two scripts, **shiftLeft.py** and **shiftRight.py**, that expect a bit string as an input. The script **shiftLeft** shifts the bits in its input *one* place to the left, wrapping the leftmost bit to the rightmost position.

The script **shiftRight** performs the inverse operation. Each script prints the resulting string.

- 6 Use the strategy of the decimal to binary conversion and the bit shift left operation defined in Project 5 to code a new encryption algorithm. The algorithm should add 1 to each character's numeric ASCII value, convert it to a bit string, and shift the bits of this string one place to the left. A single-space character in the encrypted string separates the resulting bit strings.
- 7 Write a script that decrypts a message coded by the method used in Project 6.
- 8 Write a script named **copyfile.py**. This script should prompt the user for the names of two text files. The contents of the first file should be input and written to the second file.
- 9 Write a script named **dif.py**. This script should prompt the user for the names of two text files and compare the contents of the two files to see if they are the same. If they are, the script should simply output **"Yes"**. If they are not, the script should output **"No"**, followed by the first lines of each file that differ from each other. The input loop should read and compare lines from each file. The loop should **break** as soon as a pair of different lines is found.
- 10 The Payroll Department keeps a list of employee information for each pay period in a text file. The format of each line of the file is the following:

```
<last name> <hourly wage> <hours worked>
```

Write a program that inputs a filename from the user and prints to the terminal a report of the wages paid to the employees for the given period. The report should be in tabular format with the appropriate header. Each line should contain an employee's name, the hours worked, and the wages paid for that period.

[CHAPTER] 5 Lists and Dictionaries

After completing this chapter, you will be able to:

- Construct lists and access items in those lists
- Use methods to manipulate lists
- Perform traversals of lists to process items in the lists
- Define simple functions that expect parameters and return values
- Construct dictionaries and access entries in those dictionaries
- Use methods to manipulate dictionaries
- Decide whether a list or a dictionary is an appropriate data structure for a given application

As data-processing problems have become more complex, computer scientists have developed data structures to help solve them. A data structure combines several data values into a unit so they can be treated as one thing. The data elements within a data structure are usually organized in a special way that allows the programmer to access and manipulate them. As you saw in Chapter 4, a string is a data structure that organizes text as a sequence of characters. In this chapter, we explore the use of two other common data structures: the list and the dictionary. A **list** allows the programmer to manipulate a sequence of data values of any types. A **dictionary** organizes data values by association with other data values rather than by sequential position.

Lists and dictionaries provide powerful ways to organize data in useful and interesting applications. In addition to exploring the use of lists and dictionaries, this chapter also introduces the definition of simple functions. These functions help to organize program code, in much the same manner as data structures help to organize data.

5.1

Lists

A list is a sequence of data values called **items** or **elements**. An item can be of any type. Here are some real-world examples of lists:

- A shopping list for the grocery store
- A to-do list
- A roster for an athletic team
- A guest list for a wedding
- A recipe, which is a list of instructions
- A text document, which is a list of lines
- The words in a dictionary
- The names in a phone book

The logical structure of a list is similar to the structure of a string. Each of the items in a list is ordered by position. Like a character in a string, each item in a list has a unique **index** that specifies its position. The index of the first item is 0, and the index of the last item is the length of the list minus 1. As sequences, lists and strings share many of the same operators, but include different sets of methods. We now examine these in detail.

5.1.1

List Literals and Basic Operators

In Python, a list is written as a sequence of data values separated by commas. The entire sequence is enclosed in square brackets (**[** and **]**). Here are some example lists:

```
[1951, 1969, 1984]           # A list of integers
['apples', 'oranges', 'cherries'] # A list of strings
[]                           # An empty list
```

You can also use other lists as elements in a list, thereby creating a list of lists. Here is one example of such a list:

```
[[5, 9], [541, 78]]
```

It is interesting that when the Python interpreter evaluates a list literal, each of the elements is evaluated as well. When an element is a number or a string, that literal is included in the resulting list. However, when the element is a variable or any other expression, its value is included in the list, as shown in the following session:

```
>>> import math
>>> x = 2
>>> [x, math.sqrt(x)]
[2, 1.4142135623730951]
>>> [x + 1]
[3]
>>>
```

You can also build lists of integers using the **range** and **list** functions introduced in Chapter 3. The next session shows the construction of two lists and their assignment to variables:

```
>>> first = [1, 2, 3, 4]
>>> second = list(range(1, 5))
>>> first
[1, 2, 3, 4]
>>> second
[1, 2, 3, 4]
>>>
```

The function **len** and the subscript operator **[]** work just as they do for strings:

```
>>> len(first)
4
>>> first[0]
1
>>> first[2:4]
[3, 4]
>>>
```

Concatenation (+) and equality (==) also work as expected for lists:

```
>>> first + [5, 6]
[1, 2, 3, 4, 5, 6]
>>> first == second
True
>>>
```

The **print** function strips the quotation marks from a string, but does not alter the look of a list:

```
>>> print("1234")
1234
>>> print([1, 2, 3, 4])
[1, 2, 3, 4]
>>>
```

To print the contents of a list without the brackets and commas, you can use a **for** loop, as follows:

```
>>> for element in [1, 2, 3, 4]:
    print(element, end=" ")

1 2 3 4
>>>
```

Finally, you can use the **in** operator to detect the presence or absence of a given element:

```
>>> 3 in [1, 2, 3]
True
>>> 0 in [1, 2, 3]
False
>>>
```

Table 5.1 summarizes these operators and functions, where **L** refers to a list.

OPERATOR OR FUNCTION	WHAT IT DOES
<code>L[<i><an integer expression></i>]</code>	Subscript used to access an element at the given index position.
<code>L[<i><start></i>:<i><end></i>]</code>	Slices for a sublist. Returns a new list.
<code>L + L</code>	List concatenation. Returns a new list consisting of the elements of the two operands.
<code>print(L)</code>	Prints the literal representation of the list.
<code>len(L)</code>	Returns the number of elements in the list.
<code>list(range(<i><upper></i>))</code>	Returns a list containing the integers in the range 0 through <i>upper</i> - 1.
<code>==, !=, <, >, <=, >=</code>	Compares the elements at the corresponding positions in the operand lists. Returns True if all the results are true, or False otherwise.
<code>for <variable> in L: <statement></code>	Iterates through the list, binding the variable to each element.
<code><any value> in L</code>	Returns True if the value is in the list or False otherwise.

[TABLE 5.1] Some operators and functions used with lists

5.1.2 Replacing an Element in a List

The examples discussed thus far might lead you to think that a list behaves more or less like a string. However, there is one huge difference. Because a string is immutable, its structure and contents cannot be changed. But a list is changeable—that is, it is **mutable**. At any point in its lifetime, elements can be inserted, removed, or replaced. The list itself maintains its identity, but its **state**—its length and its contents—can change.

The subscript operator is used to replace an element at a given position, as shown in the next session:

```
>>> example = [1, 2, 3, 4]
>>> example
[1, 2, 3, 4]
>>> example[3] = 0
>>> example
[1, 2, 3, 0]
>>>
```

Note that the subscript is used to reference the **target** of the assignment statement, which is not the list but an element's position within it. Much of list processing involves replacing each element, with the result of applying some operation to that element. We now present two examples of how this is done.

The first session shows how to replace each number in a list with its square:

```
>>> numbers = [2, 3, 4, 5]
>>> numbers
[2, 3, 4, 5]
>>> index = 0
>>> while index < len(numbers):
    numbers[index] = numbers[index] ** 2
    index += 1

>>> numbers
[4, 9, 16, 25]
>>>
```

Note that the code uses a **while** loop over the index rather than a **for** loop over the list elements, because the index is needed to access the positions for the assignments.

The next session uses the string method **split** to extract a list of the words in a sentence. These words are then converted to uppercase letters within the list:

```
>>> sentence = "This example has five words."
>>> words = sentence.split()
>>> words
['This', 'example', 'has', 'five', 'words.']
>>> index = 0
```

continued

```
>>> while index < len(words):
    words[index] = words[index].upper()
    index += 1

>>> words
['THIS', 'EXAMPLE', 'HAS', 'FIVE', 'WORDS.']
>>>
```

You can also replace a sublist of elements within a list by slicing. The slice operator appears on the left side of the assignment operator, while the sublist of replacements appears on the right. The next example replaces the first three elements of a list with new ones:

```
>>> numbers = list(range(6))
>>> numbers
[0, 1, 2, 3, 4, 5]
>>> numbers[0:3] = [11, 12, 13]
>>> numbers
[11, 12, 13, 3, 4, 5]
>>>
```

5.1.3 List Methods for Inserting and Removing Elements

The **list** type includes several methods for inserting and removing elements. These methods are summarized in Table 5.2, where **L** refers to a list. To learn more about these methods, enter **help(list)** in a Python shell.

LIST METHOD	WHAT IT DOES
L.append(element)	Adds element to the end of L .
L.extend(aList)	Adds the elements of aList to the end of L .
L.insert(index, element)	Inserts element at index if index is less than the length of L . Otherwise, inserts element at the end of L .
L.pop()	Removes and returns the element at the end of L .
L.pop(index)	Removes and returns the element at index .

[TABLE 5.2] List methods for inserting and removing elements

The method **insert** expects an integer index and the new element as arguments. When the index is less than the length of the list, this method places the new element before the existing element at that index, after shifting elements to the right by one position. At the end of the operation, the new element occupies the given index position. When the index is greater than or equal to the length of the list, the new element is added to the end of the list. The next session shows **insert** in action:

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.insert(1, 10)
>>> example
[1, 10, 2]
>>> example.insert(3, 25)
>>> example
[1, 10, 2, 25]
>>>
```

The method **append** is a simplified version of **insert**. The method **append** expects just the new element as an argument and adds the new element to the end of the list. The method **extend** performs a similar operation, but adds the elements of its list argument to the end of the list. The next session shows the difference between **append** and **extend**:

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.append(10)
>>> example
[1, 2, 10]
>>> example.extend([11, 12, 13])
>>> example
[1, 2, 10, 11, 12, 13]
>>>
```

The method **pop** is used to remove an element at a given position. If the position is not specified, **pop** removes and returns the last element. If the position is specified, **pop** removes the element at that position and returns it. In that case,

the elements that followed the removed element are shifted one position to the left. The next session removes the last and first elements from the example list:

```
>>> example
[1, 2, 10, 11, 12, 13]
>>> example.pop()
13
>>> example
[1, 2, 10, 11, 12]
>>> example.pop(0)
1
>>> example
[2, 10, 11, 12]
>>>
```

5.1.4 Searching a List

After elements have been added to a list, a program can search for a given element. The **in** operator determines an element's presence or absence, but programmers often are more interested in the position of an element if it is found (for replacement, removal, or other use). Unfortunately, the **list** type does not include the convenient **find** method that is used with strings. Recall that **find** returns either the index of the given substring in a string or -1 if the substring is not found. Instead of **find**, you must use the method **index** to locate an element's position in a list. It is unfortunate that **index** raises an error when the target element is not found. To guard against this unpleasant consequence, you must first use the **in** operator to test for presence and then the **index** method if this test returns **True**. The next code segment shows how this is done for an example list and target element:

```
aList = [34, 45, 67]
target = 45
if target in aList:
    print(aList.index(target))
else:
    print(-1)
```

5.1.5 Sorting a List

Although a list's elements are always ordered by position, it is possible to impose a **natural ordering** on them as well. In other words, you can arrange some elements in numeric or alphabetical order. A list of numbers in ascending order and a list of names in alphabetical order are sorted lists. When the elements can be related by comparing them for less than and greater than as well as equality, they can be sorted. The **list** method **sort** mutates a list by arranging its elements in ascending order. Here is an example of its use:

```
>>> example = [4, 2, 10, 8]
>>> example
[4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

5.1.6 Mutator Methods and the Value None

All of the functions and methods examined in previous chapters return a value that the caller can then use to complete its work. Mutable objects (such as lists) have some methods devoted entirely to modifying the internal state of the object. Such methods are called **mutators**. Examples are the **list** methods **insert**, **append**, **extend**, and **sort**. Because a change of state is all that is desired, a mutator method usually returns no value of interest to the caller. Python nevertheless automatically returns the special value **None** even when a method does not explicitly return a value. We mention this now only as a warning against the following type of error. Suppose you forget that **sort** mutates a list, and instead you mistakenly think that it builds and returns a new, sorted list and leaves the original list unsorted. Then, you might write code like the following to obtain what you think is the desired result:

```
>>> aList = aList.sort()
```

Unfortunately, after the list object is sorted, this assignment has the result of setting the variable **aList** to the value **None**. The next **print** statement shows that the reference to the list object is lost:

```
>>> print(aList)
None
```

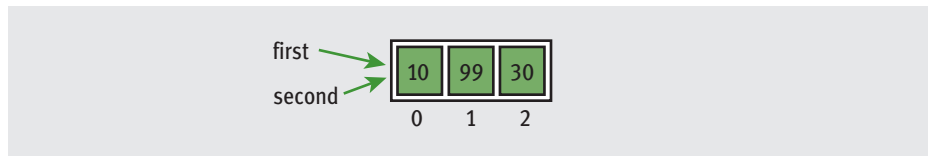
Later in this book, you will learn how to make something useful out of **None**.

5.1.7 Aliasing and Side Effects

As you learned earlier, numbers and strings are immutable. That is, you cannot change their internal structure. However, because lists are mutable, you can replace, insert, or remove elements. The mutable property of lists leads to some interesting phenomena, as shown in the following session:

```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
>>> first[1] = 99
>>> first
[10, 99, 30]
>>> second
[10, 99, 30]
>>>
```

In this example, a single list object is created and modified using the subscript operator. When the second element of the list named **first** is replaced, the second element of the list named **second** is replaced also. This type of change is what is known as a **side effect**. This happens because after the assignment **second = first**, the variables **first** and **second** refer to the exact same list object. They are **aliases** for the same object, as shown in Figure 5.1. This phenomenon is known as **aliasing**.



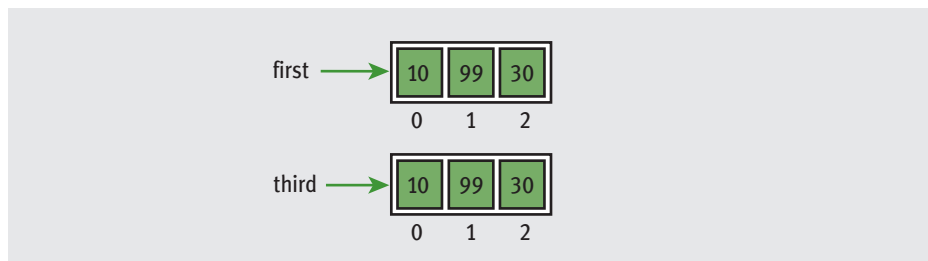
[FIGURE 5.1] Two variables refer to the same list object

If the data are immutable strings, aliasing can save on memory. But as you might imagine, aliasing is not always a good thing when side effects are possible. Assignment creates an alias to the same object rather than a reference to a copy of the object. To prevent aliasing, you can create a new object and copy the contents of the original to it, as shown in the next session:

```
>>> third = []
>>> for element in first:
>>>     third.append(element)

>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
>>> first[1] = 100
>>> first
[10, 100, 30]
>>> third
[10, 99, 30]
>>>
```

The variables **first** and **third** refer to two different list objects, although their contents are initially the same, as shown in Figure 5.2. The important point is that they are not aliases, so you don't have to be concerned about side effects.



[FIGURE 5.2] Two variables refer to different list objects

A simpler way to copy a list is to use a slice over all of the positions, as follows:

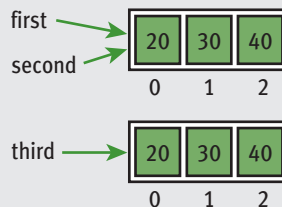
```
>>> third = first[:]
```

5.1.8 Equality: Object Identity and Structural Equivalence

Occasionally, programmers need to see whether two variables refer to the exact same object or to different objects. For example, you might want to determine whether one variable is an alias for another. The `==` operator returns **True** if the variables are aliases for the same object. Unfortunately, `==` also returns **True** if the contents of two different objects are the same. The first relation is called **object identity**, whereas the second relation is called **structural equivalence**. The `==` operator has no way of distinguishing between these two types of relations.

Python's `is` operator can be used to test for object identity. It returns **True** if the two operands refer to the exact same object, and it returns **False** if the operands refer to distinct objects (even if they are structurally equivalent). The next session shows the difference between `==` and `is`, and Figure 5.3 depicts the objects in question.

```
>>> first = [20, 30, 40]
>>> second = first
>>> third = [20, 30, 40]
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
>>>
```



[FIGURE 5.3] Three variables and two distinct list objects

5.1.9

Example: Using a List to Find the Median of a Set of Numbers

Researchers who do quantitative analysis are often interested in the **median** of a set of numbers. For example, the U.S. Government often gathers data to determine the median family income. Roughly speaking, the median is the value that is less than half the numbers in the set and greater than the other half. If the number of values in a list is odd, the median of the list is the value at the midpoint when the set of numbers is sorted; otherwise, the median is the average of the two values surrounding the midpoint. Thus, the median of the list [1, 3, 3, 5, 7] is 3, and the median of the list [1, 2, 4, 4] is also 3. The following script inputs a set of numbers from a text file and prints their median:

```
"""
File: median.py
Prints the median of a set of numbers in a file.
"""

fileName = input("Enter the filename: ")
f = open(fileName, 'r')

# Input the text, convert it to numbers, and
# add the numbers to a list
numbers = []
for line in f:
    words = line.split()
    for word in words:
        numbers.append(float(word))

# Sort the list and print the number at its midpoint
numbers.sort()
midpoint = len(numbers) // 2
print("The median is", end=" ")
if len(numbers) % 2 == 1:
    print(numbers[midpoint])
else:
    print((numbers[midpoint] + numbers[midpoint - 1]) / 2)
```

Note that the input process is the most complex part of this script. An accumulator list, **numbers**, is set to the empty list. The **for** loop reads each line of text and extracts a list of words from that line. The nested **for** loop traverses this list to convert each word to a number. The **list** method **append** then adds each

number to the end of **numbers**, the accumulator list. The remaining lines of code locate the median value. When run with an input file whose contents are

```
3 2 7
8 2 1
5
```

the script produces the following output:

```
The median is 3.0
```

5.1.10 Tuples

A **tuple** is a type of sequence that resembles a list, except that, unlike a list, a tuple is immutable. You indicate a tuple literal in Python by enclosing its elements in parentheses instead of square brackets. The next session shows how to create several tuples:

```
>>> fruits = ("apple", "banana")
>>> fruits
('apple', 'banana')
>>> meats = ("fish", "poultry")
>>> meats
('fish', 'poultry')
>>> food = meats + fruits
>>> food
('fish', 'poultry', 'apple', 'banana')
>>> veggies = ["celery", "beans"]
>>> tuple(veggies)
('celery', 'beans')
```

You can use most of the operators and functions used with lists in a similar fashion with tuples. For the most part, anytime you foresee using a list whose structure will not change, you can, and should, use a tuple instead. For example, the set of vowels and the set of punctuation marks in a text-processing application could be represented as tuples of strings.

5.1 Exercises

- 1 Assume that the variable **data** refers to the list `[5, 3, 7]`. Write the values of the following expressions:
 - a `data[2]`
 - b `data[-1]`
 - c `len(data)`
 - d `data[0:2]`
 - e `0 in data`
 - f `data + [2, 10, 5]`
 - g `tuple(data)`
- 2 Assume that the variable **data** refers to the list `[5, 3, 7]`. Write the expressions that perform the following tasks:
 - a Replace the value at position 0 in **data** with that value's negation.
 - b Add the value 10 to the end of **data**.
 - c Insert the value 22 at position 2 in **data**.
 - d Remove the value at position 1 in **data**.
 - e Add the values in the list **newData** to the end of **data**.
 - f Locate the index of the value 7 in **data**, safely.
 - g Sort the values in **data**.
- 3 What is a mutator method? Explain why mutator methods usually return the value **None**.
- 4 Write a loop that accumulates the sum of all of the numbers in a list named **data**.
- 5 Assume that **data** refers to a list of numbers, and **result** refers to an empty list. Write a loop that adds the nonzero values in **data** to the **result** list.
- 6 Write a loop that replaces each number in a list named **data** with its absolute value.
- 7 Describe the costs and benefits of aliasing, and explain how it can be avoided.
- 8 Explain the difference between structural equivalence and object identity.

5.2 Defining Simple Functions

Thus far, our programs have consisted of short code segments or scripts. Some of these have used built-in functions to do useful work. Some of our scripts might also be useful enough to package as functions to be used in other scripts. Moreover, defining our own functions allows us to organize our code in existing scripts more effectively. This section provides a brief overview of how to do this. We'll examine functions in more detail in Chapter 6.

5.2.1 The Syntax of Simple Function Definitions

Most of the functions used thus far expect one or more arguments and return a value. Let's define a function that expects a number as an argument and returns the square of that number. First, we consider how the function will be used. Its name is **square**, so you can call it like this:

```
>>> square(2)
4
>>> square(6)
36
>>>
```

The definition of this function consists of a header and a body. Here is the code:

```
def square(x):
    """Returns the square of x. """
    return x * x
```

The function's header contains the function's name and a parenthesized list of argument names. The function's body contains the statements that execute when the function is called. Our function contains a single **return** statement, which simply returns the result of multiplying its argument, named **x**, by itself. Note that the argument name, also called a parameter, behaves just like a variable in the body of the function. This variable does not receive an initial value until the function is called. For example, when the function is called with the argument 6, the parameter **x** will have the value 6 in the function's body.

Our function also contains a docstring. This string contains information about what the function does. It is displayed in the shell when the programmer enters **help(square)**.

A function can be defined in a Python shell, but it is more convenient to define it in an IDLE window, where it can be saved to a file. Loading the window into the shell then loads the function definition as well. Like variables, functions generally must be defined in a script before they are actually called in that same script.

Our next example function computes the average value in a list of numbers. The function might be used as follows:

```
>>> average([1, 3, 5, 7])
4.0
```

Here is the code for the function's definition:

```
def average(list):
    """Returns the average of the numbers in list."""
    sum = 0
    for number in list:
        sum += number
    return sum / len(list)
```

The syntax of a function definition contains a header and a body. The header consists of the reserved word **def**, followed by the function's name, followed by a parenthesized list of parameters and a colon, as follows:

```
def <function name>(<parameter-1>, ..., <parameter-n>):
    <body>
```

The function's body contains one or more statements.

5.2.2 Parameters and Arguments

A parameter is the name used in the function definition for an argument that is passed to the function when it is called. For now, the number and positions of the arguments of a function call should match the number and positions of the parameters in that function's definition. Some functions expect no arguments, so they are defined with no parameters.

5.2.3 The `return` Statement

The programmer places a **return** statement at each exit point of a function when that function should explicitly return a value. The syntax of the **return** statement is the following:

```
return <expression>
```

Upon encountering a **return** statement, Python evaluates the expression and immediately transfers control back to the caller of the function. The value of the expression is also sent back to the caller. If a function contains no **return** statement, Python transfers control to the caller after the last statement in the function's body is executed, and the special value **None** is automatically returned.

5.2.4 Boolean Functions

A **Boolean function** usually tests its argument for the presence or absence of some property. The function returns **True** if the property is present, or **False** otherwise. The next example shows the use and definition of the Boolean function **odd**, which tests a number to see whether it is odd.

```
>>> odd(5)
True
>>> odd(6)
False

def odd(x):
    """Returns True if x is odd or False otherwise."""
    if x % 2 == 1:
        return True
    else:
        return False
```

Note that this function has two possible exit points, in either of the alternatives within the **if/else** statement.

5.2.5 Defining a `main` Function

In scripts that include the definitions of several cooperating functions, it is often useful to define a special function named `main` that serves as the entry point for the script. This function usually expects no arguments and returns no value. Its sole purpose is to take inputs, process them by calling other functions, and print the results. The definition of the `main` function and the other function definitions can appear in no particular order in the script, as long as `main` is called at the very end of the script.

The next example shows a complete script that is organized in the manner just described. The `main` function prompts the user for a number, calls the `square` function to compute its square, and prints the result. You can define the `main` and the `square` functions in any order. When Python loads this module, the code for both function definitions is loaded and compiled, but not executed. Note that `main` is then called as the last step in the script. This has the effect of transferring control to the first instruction in the `main` function's definition. When `square` is called from `main`, control is transferred from `main` to the first instruction in `square`. When a function completes execution, control returns to the next instruction in the caller's code.

```
"""
File: computesquare.py
Illustrates the definition of a main function.
"""

def main():
    """The main function for this script."""
    number = float(input("Enter a number: "))
    result = square(number)
    print("The square of", number, "is", result)

def square(x):
    """Returns the square of x."""
    return x * x

# The entry point for program execution
main()
```

Like all scripts, the preceding script can be run from IDLE, imported into the shell, or run from a terminal command prompt. We will start defining and using a `main` function in most of our case studies from this point forward.

5.2 Exercises

- 1 What roles do the parameters and the **return** statement play in a function definition?
- 2 Define a function named **even**. This function expects a number as an argument and returns **True** if the number is divisible by 2, or it returns **False** otherwise. (*Hint*: a number is evenly divisible by 2 if the remainder is 0.)
- 3 Use the function **even** to simplify the definition of the function **odd** presented in this section.
- 4 Define a function named **sum**. This function expects two numbers, named **low** and **high**, as arguments. The function computes and returns the sum of all of the numbers between **low** and **high**, inclusive.
- 5 What is the purpose of a **main** function?

5.3 Case Study: Generating Sentences

Can computers write poetry? We'll attempt to answer that question in this case study by giving a program a few words to play with.

5.3.1 Request

Write a program that generates sentences.

5.3.2 Analysis

Sentences in any language have a structure defined by a set of **grammar rules**. They also include a set of words from the **vocabulary** of the language. The vocabulary of a language like English consists of many thousands of words, and the grammar rules are quite complex. For the sake of simplicity, our program will generate sentences from a simplified subset of English. The vocabulary will consist of sample words from several parts of speech, including nouns, verbs, articles, and prepositions. From these words, you can build noun phrases, prepositional phrases, and verb phrases. From these constituent phrases, you can build sentences. For example, the sentence, "The girl hit the ball with the bat," contains

three noun phrases, one verb phrase, and one prepositional phrase. Table 5.3 summarizes the grammar rules for our subset of English.

PHRASE	ITS CONSTITUENTS
Sentence	Noun phrase + Verb phrase
Noun phrase	Article + Noun
Verb phrase	Verb + Noun phrase + Prepositional phrase
Prepositional phrase	Preposition + Noun phrase

[TABLE 5.3] The grammar rules for the sentence generator

The rule for *Noun phrase* says that it is an *Article* followed by (+) a *Noun*. Thus, a possible noun phrase is “the bat.” Note that some of the phrases in the left column of Table 5.3 also appear in the right column as constituents of other phrases. Although this grammar is much simpler than the complete set of rules for English grammar, you should still be able to generate sentences with quite a bit of structure.

The program will prompt the user for the number of sentences to generate. The proposed user interface follows:

```
> python generator.py
Enter the number of sentences: 3
THE BOY HIT THE BAT WITH A BOY
THE BOY HIT THE BALL BY A BAT
THE BOY SAW THE GIRL WITH THE GIRL

> python generator.py
Enter the number of sentences: 2
A BALL HIT A GIRL WITH THE BAT
A GIRL SAW THE BAT BY A BOY
```

5.3.3 Design

Of the many ways to solve the problem in this case study, perhaps the simplest is to assign the task of generating each phrase to a separate function. Each function builds and returns a string that represents its phrase. This string contains words drawn from the parts of speech and also from other phrases. When a function needs an individual word, it is selected at random from the words in that part of

speech. When a function needs another phrase, it calls another function to build that phrase. The results, all strings, are concatenated with spaces and returned.

The function for *Sentence* is the easiest. It just calls the functions for *Noun phrase* and *Verb phrase* and concatenates the results, as in the following:

```
def sentence():
    """Builds and returns a sentence."""
    return nounPhrase() + " " + verbPhrase() + "."
```

The function for *Noun phrase* picks an article and a noun at random from the vocabulary, concatenates them, and returns the result. We assume that the variables **articles** and **nouns** refer to collections of these parts of speech, and develop these later in the design. The function **random.choice** returns a random element from such a collection.

```
def nounPhrase():
    """Builds and returns a noun phrase."""
    return random.choice(articles) + " " + random.choice(nouns)
```

The design of the remaining two phrase-structure functions is similar.

The **main** function drives the program with a count-controlled loop:

```
def main():
    """Allows the user to input the number of sentences
    to generate."""
    number = int(input("Enter the number of sentences: "))
    for count in range(number):
        print(sentence())
```

The variables **articles** and **nouns** used in the program's functions refer to the collections of actual words belonging to these two parts of speech. Two other collections, named **verbs** and **prepositions**, also will be used. The data structure used to represent a collection of words should allow the program to pick one word at random. Because the data structure does not change during the course of the program, you can use a tuple of strings. Four tuples serve as a common pool of data for the functions in the program, and are initialized before the functions are defined.

5.3.4 Implementation (Coding)

When functions use a common pool of data, you should define or initialize the data before the functions are defined. Thus, the variables for the data are initialized just below the **import** statement.

```
"""
Program: generator.py
Author: Ken
Generates and displays sentences using simple grammar
and vocabulary. Words are chosen at random.
"""

import random

articles = ("A", "THE")

nouns = ("BOY", "GIRL", "BAT", "BALL",)

verbs = ("HIT", "SAW", "LIKED")

prepositions = ("WITH", "BY")

def sentence():
    """Builds and returns a sentence."""
    return nounPhrase() + " " + verbPhrase()

def nounPhrase():
    """Builds and returns a noun phrase."""
    return random.choice(articles) + " " + random.choice(nouns)

def verbPhrase():
    """Builds and returns a verb phrase."""
    return random.choice(verbs) + " " + nounPhrase() + " " + \
        prepositionalPhrase()

def prepositionalPhrase():
    """Builds and returns a prepositional phrase."""
    return random.choice(prepositions) + " " + nounPhrase()

def main():
    """Allows the user to input the number of sentences
    to generate."""
    number = int(input("Enter the number of sentences: "))
    for count in range(number):
        print(sentence())

main()
```

5.3.5 Testing

Poetry it's not, but testing is still important. The functions developed in this case study can be tested in a bottom-up manner. To do so, you must initialize the data first. Then you can run the lowest-level function, **nounPhrase**, immediately to check its results, and you can work up to sentences from there.

On the other hand, testing can also follow the design, which took a top-down path. You might start by writing headers for all of the functions and simple **return** statements that return the function's names. Then you can complete the code for the **sentence** function first, test it, and proceed downward from there. The wise programmer can also mix bottom-up and top-down testing as needed.

5.4 Dictionaries

Lists organize their elements by position. This mode of organization is useful when you want to locate the first element, the last element, or visit each element in a sequence. However, in some situations, the position of a datum in a structure is irrelevant; we're interested in its association with some other element in the structure. For example, you might want to look up Ethan's phone number but don't care where that number is in the phone book.

A dictionary organizes information by **association**, not position. For example, when you use a dictionary to look up the definition of "mammal," you don't start at page 1; instead, you turn directly to the words beginning with "M." Phone books, address books, encyclopedias, and other reference sources also organize information by association. In computer science, data structures organized by association are also called **tables** or **association lists**. In Python, a **dictionary** associates a set of **keys** with data values. For example, the keys in *Webster's Dictionary* comprise the set of words, whereas the associated data values are their definitions. In this section, we examine the use of dictionaries in data processing.

5.4.1 Dictionary Literals

A Python dictionary is written as a sequence of key/value pairs separated by commas. These pairs are sometimes called **entries**. The entire sequence of entries is

enclosed in curly braces (`{` and `}`). A colon (`:`) separates a key and its value. Here are some example dictionaries:

A phone book: `{ 'Savannah': '476-3321', 'Nathaniel': '351-7743' }`

Personal information: `{ 'Name': 'Molly', 'Age': 18 }`

You can even create an empty dictionary—that is, a dictionary that contains no entries. You would create an empty dictionary in a program that builds a dictionary from scratch. Here is an example of an empty dictionary:

```
{}
```

The keys in a dictionary can be data of any immutable types, including other data structures, although keys normally are strings or integers. The associated values can be of any types. Although the entries may appear to be ordered in a dictionary, this ordering is not significant, and the programmer should not rely on it.

5.4.2 Adding Keys and Replacing Values

You add a new key/value pair to a dictionary by using the subscript operator `[]`. The form of this operation is the following:

```
<a dictionary>[<a key>] = <a value>
```

The next code segment creates an empty dictionary and adds two new entries:

```
>>> info = {}
>>> info["name"] = "Sandy"
>>> info["occupation"] = "hacker"
>>> info
{'name': 'Sandy', 'occupation': 'hacker'}
```

The subscript is also used to replace a value at an existing key, as follows:

```
>>> info["occupation"] = "manager"
>>> info
{'name': 'Sandy', 'occupation': 'manager'}
>>>
```

Here is a case of the same operation used for two different purposes, insertion of a new entry and modification of an existing entry. As a rule, when the key is absent from the dictionary, it and its value are inserted; when the key already exists, its associated value is replaced.

5.4.3 Accessing Values

You can also use the subscript to obtain the value associated with a key. However, if the key is not present in the dictionary, Python raises an error. Here are some examples, using the **info** dictionary, which was set up earlier:

```
>>> info["name"]
'Sandy'
>>> info["job"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'job'
>>>
```

If the existence of a key is uncertain, the programmer can test for it using the dictionary method **has_key**, but a far easier strategy is to use the method **get**. This method expects two arguments, a possible key and a default value. If the key is in the dictionary, the associated value is returned. However, if the key is absent, the default value passed to **get** is returned. Here is an example of the use of **get** with a default value of **None**:

```
>>> print(info.get("job", None))
None
>>>
```

5.4.4 Removing Keys

To delete an entry from a dictionary, one removes its key using the method **pop**. This method expects a key and an optional default value as arguments. If the key is in the dictionary, it is removed, and its associated value is returned. Otherwise, the default value is returned. If **pop** is used with just one argument, and this key is absent from the dictionary, Python raises an error. The next session attempts to remove two keys and prints the values returned:

```
>>> print(info.pop("job", None))
None
>>> print(info.pop("occupation"))
manager
>>> info
{'name': 'Sandy'}
>>>
```

5.4.5 Traversing a Dictionary

When a **for** loop is used with a dictionary, the loop's variable is bound to each key in an unspecified order. The next code segment prints all of the keys and their values in our **info** dictionary:

```
for key in info:
    print(key, info[key])
```

Alternatively, you could use the dictionary method **items()** to access a list of the dictionary's entries. The next session shows a run of this method with a dictionary of grades:

```
>>> grades = {90:"A", 80:"B", 70:"C"}
>>> grades.items()
[(80, 'B'), (90, 'A'), (70, 'C')]
```

Note that the entries are represented as tuples within the list. A tuple of variables can then access the key and value of each entry in this list within a **for** loop:

```
for (key, value) in grades.items():  
    print(key, value)
```

On each pass through the loop, the variables **key** and **value** within the tuple are assigned the key and value of the current entry in the list.

If a special ordering of the keys is needed, you can obtain a list of keys using the **keys** method and process this list to rearrange the keys. For example, you can sort the list and then traverse it to print the entries of the dictionary in alphabetical order:

```
theKeys = list(info.keys())  
theKeys.sort()  
for key in theKeys:  
    print(key, info[key])
```

To see the complete documentation for dictionaries, you can run **help(dict)** at a shell prompt. Table 5.4 summarizes the commonly used dictionary operations, where **d** refers to a dictionary.

DICTIONARY OPERATION	WHAT IT DOES
len(d)	Returns the number of entries in d .
aDict[key]	Used for inserting a new key, replacing a value, or obtaining a value at an existing key.
d.get(key [, default])	Returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
d.pop(key [, default])	Removes the key and returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
list(d.keys())	Returns a list of the keys.
list(d.values())	Returns a list of the values.

continued

DICTIONARY OPERATION	WHAT IT DOES
<code>list(d.items())</code>	Returns a list of tuples containing the keys and values for each entry.
<code>d.has_key(key)</code>	Returns True if the key exists or False otherwise.
<code>d.clear()</code>	Removes all the keys.
<code>for key in d:</code>	key is bound to each key in d in an unspecified order.

[TABLE 5.4] Some commonly used dictionary operations

5.4.6 Example: The Hexadecimal System Revisited

In Chapter 4, we discussed a method for converting numbers quickly between the binary and the hexadecimal systems. Now let's develop a Python function that uses that method to convert a hexadecimal number to a binary number. The algorithm visits each digit in the hexadecimal number, selects the corresponding four bits that represent that digit in binary, and adds these bits to a result string. You could express this selection process with a complex **if/else** statement, but there is an easier way. If you maintain the set of associations between hexadecimal digits and binary digits in a dictionary, then you can just look up each hexadecimal digit's binary equivalent with a primitive operation. Such a dictionary is sometimes called a **lookup table**. Here is the definition of the lookup table required for hex-to-binary conversions:

```
hexToBinaryTable = {'0':'0000', '1':'0001', '2':'0010',
                    '3':'0011', '4':'0100', '5':'0101',
                    '6':'0110', '7':'0111', '8':'1000',
                    '9':'1001', 'A':'1010', 'B':'1011',
                    'C':'1100', 'D':'1101', 'E':'1110',
                    'F':'1111'}
```

The function itself, named **convert**, is simple. It expects two parameters: a string representing the number to be converted and a table of associations of digits. Here is the code for the function, followed by a sample session:

```
def convert(number, table):
    """Builds and returns the base two representation of
    number."""
    binary = ''
    for digit in number:
        binary = table[digit] + binary
    return binary

>>> convert("35A", hexToBinaryTable)
'101001010011'
```

Note that you pass **hexToBinaryTable** as an argument to the function. The function then uses the associations in this particular table to perform the conversion. The function would serve equally well for conversions from octal to binary, provided that you set up and pass it an appropriate lookup table.

5.4.7 Example: Finding the Mode of a List of Values

The **mode** of a list of values is the value that occurs most frequently. The following script inputs a list of words from a text file and prints their mode. The script uses a list and a dictionary. The list is used to obtain the words from the file, as in earlier examples. The dictionary associates each unique word with the number of its occurrences in the list. The script also uses the function **max**, first introduced in Chapter 3, to compute the maximum of two values. When used with a single list argument, **max** returns the largest value contained therein. Here is the code for the script:

```
fileName = input("Enter the filename: ")
f = open(fileName, 'r')

# Input the text, convert its words to uppercase, and
# add the words to a list
words = []
for line in f:
    wordsInLine = line.split()
    for word in wordsInLine:
        words.append(word.upper())
```

continued


```

# Obtain the set of unique words and their
# frequencies, saving these associations in
# a dictionary
theDictionary = {}
for word in words:
    number = theDictionary.get(word, None)
    if number == None:
        # word entered for the first time
        theDictionary[word] = 1
    else:
        # word already seen, increment its number
        theDictionary[word] = number + 1

# Find the mode by obtaining the maximum value
# in the dictionary and determining its key
theMaximum = max(theDictionary.values())
for key in theDictionary:
    if theDictionary[key] == theMaximum:
        print("The mode is", key)
        break

```

5.4 Exercises

- 1 Give three examples of real-world objects that behave like a dictionary.
- 2 Assume that the variable **data** refers to the dictionary **{"b":20, "a":35}**. Write the values of the following expressions:
 - a **data["a"]**
 - b **data.get("c", None)**
 - c **len(data)**
 - d **data.keys()**
 - e **data.values()**
 - f **data.pop("b")**
 - g **data** # After the pop above
- 3 Assume that the variable **data** refers to the dictionary **{"b":20, "a":35}**. Write the expressions that perform the following tasks:
 - a Replace the value at the key **"b"** in **data** with that value's negation.
 - b Add the key/value pair **"c":40** to **data**.
 - c Remove the value at key **"b"** in **data**, safely.
 - d Print the keys in **data** in alphabetical order.

5.5

Case Study: Nondirective Psychotherapy

In the early 1960s, the MIT computer scientist Joseph Weizenbaum developed a famous program called **doctor** that could converse with the computer user, mimicking a nondirective style of psychotherapy. The doctor in this kind of therapy is essentially a good listener who responds to the patient's statements by rephrasing them or indirectly asking for more information. To illustrate the use of data structures, we develop a drastically simplified version of this program.

5.5.1

Request

Write a program that emulates a nondirective psychotherapist.

5.5.2

Analysis

Figure 5.4 shows the program's interface as it changes throughout a sequence of exchanges with the user.

```
Good morning, I hope you are well today.  
What can I do for you?  
  
>> My mother and I don't get along  
Why do you say that your mother and you don't get along  
  
>> she always favors my sister  
You seem to think that she always favors your sister  
  
>> my dad and I get along fine  
Can you explain why your dad and you get along fine  
  
>> he helps me with my homework  
Please tell me more  
  
>> quit  
Have a nice day!
```

[FIGURE 5.4] A session with the doctor program

When the user enters a statement, the program responds in one of two ways:

- 1 With a randomly chosen hedge, such as “Please tell me more.”
- 2 By changing some key words in the user’s input string and appending this string to a randomly chosen qualifier. Thus, to “My teacher always plays favorites,” the program might reply, “Why do you say that your teacher always plays favorites?”

5.5.3 Design

The program consists of a set of collaborating functions that share a common data pool.

Two of the data sets are the hedges and the qualifiers. Because these collections do not change and their elements must be selected at random, you can use tuples to represent them. Their names, of course, are **hedges** and **qualifiers**.

The other set of data consists of mappings between first-person pronouns and second-person pronouns. For example, when the program sees “I” in a patient’s input, it should respond with a sentence containing “you.” The best type of data structure to hold these correlations is a dictionary. This dictionary is named **replacements**.

The **main** function displays a greeting, displays a prompt, and waits for user input. The following is pseudocode for the main loop:

```
output a greeting to the patient
while True
    prompt for and input a string from the patient
    if the string equals “Quit”
        output a sign-off message to the patient
        break
    call another function to obtain a reply to this string
    output the reply to the patient
```

Our therapist might not be an expert, but there is no charge for its services. What’s more, our therapist seems willing to go on forever. However, if the patient must quit to do something else, she can do so by typing quit to end the program.

The **reply** function expects the patient’s string as an argument and returns another string as the reply. This function implements the two strategies for making replies suggested in the analysis phase. A quarter of the time a hedge is warranted. Otherwise, the function constructs its reply by changing the persons in the patient’s input and appending the result to a randomly selected qualifier. The

reply function calls yet another function, **changePerson**, to perform the complex task of changing persons.

```
def reply(sentence):
    """Builds and returns a reply to the sentence."""
    probability = random.randint(1, 4)
    if probability == 1:
        return random.choice(hedges)
    else:
        return random.choice(qualifiers) + changePerson(sentence)
```

The **changePerson** function extracts a list of words from the patient's string. It then builds a new list wherein any pronoun key in the replacements dictionary is replaced by its pronoun/value. This list is then converted back to a string and returned.

```
def changePerson(sentence):
    words = sentence.split()
    replyWords = []
    for word in words:
        replyWords.append(replacements.get(word, word))
    return " ".join(replyWords)
```

Note that the attempt to get a replacement from the **replacements** dictionary either succeeds and returns an actual replacement pronoun, or the attempt fails and returns the original word. The string method **join** glues together the words from the **replyWords** list with a space character as a separator.

5.5.4 Implementation (Coding)

The structure of this program is similar to that of the sentence generator developed in the first case study of this chapter. The three data structures are initialized near the beginning of the program, and they never change. The three functions collaborate in a straightforward manner. Here is the code:

```
"""
Program: doctor.py
Author: Ken
Conducts an interactive session of nondirective psychotherapy.
"""
```

continued

```

import random

hedges = ("Please tell me more.",
          "Many of my patients tell me the same thing.",
          "Please continue.")

qualifiers = ("Why do you say that ",
              "You seem to think that ",
              "Can you explain why ")

replacements = {"I":"you", "me":"you", "my":"your",
               "we":"you", "us":"you", "mine":"yours"}

def reply(sentence):
    """Builds and returns a reply to the sentence."""
    probability = random.randint(1, 4)
    if probability == 1:
        return random.choice(hedges)
    else:
        return random.choice(qualifiers) + changePerson(sentence)

def changePerson(sentence):
    """Replaces first person pronouns with second person
    pronouns."""
    words = sentence.split()
    replyWords = []
    for word in words:
        replyWords.append(replacements.get(word, word))
    return " ".join(replyWords)

def main():
    """Handles the interaction between patient and doctor."""
    print("Good morning, I hope you are well today.")
    print("What can I do for you?")
    while True:
        sentence = input("\n>> ")
        if sentence.upper() == "QUIT":
            print("Have a nice day!")
            break
        print(reply(sentence))

main()

```

5.5.5 Testing

As in the sentence-generator program, the functions in this program can be tested in a bottom-up or a top-down manner. As you will see, the program's replies break down when the user addresses the therapist in the second person, uses contractions (for example, I'm and I'll), and in many other ways. As you'll see in the Projects at the end of this chapter, with a little work you can make the replies more realistic.

Summary

- A list is a sequence of zero or more elements. The elements can be of any types. The **len** function returns the number of elements in its list argument. Each element occupies a position in the list. The positions range from 0 to the length of the list minus 1.
- Lists can be manipulated with many of the operators used with strings, such as the subscript, concatenation, comparison, and **in** operators. Slicing a list returns a sublist.
- The list is a mutable data structure. An element can be replaced with a new element, added to the list, or removed from the list. Replacement uses the subscript operator. The **list** type includes several methods for insertion and removal of elements.
- The method **index** returns the position of a target element in a list. If the element is not in the list, an error is raised.
- The elements of a list can be arranged in ascending order by calling the **sort** method.
- Mutator methods are called to change the state of an object. These methods usually return the value **None**. This value is automatically returned by any function or method that does not have a **return** statement.
- Assignment of one variable to another variable causes both variables to refer to the same data object. When two or more variables refer to the same data object, they are aliases. When that data value is a mutable object such as a list, side effects can occur. A side effect is an unexpected change to the contents of a data object. To prevent side effects, avoid aliasing by assigning a copy of the original data object to the new variable.

- A tuple is quite similar to a list, but has an immutable structure.
- A function definition consists of a header and a body. The header contains the function's name and a parenthesized list of argument names. The body consists of a set of statements.
- The **return** statement returns a value from a function definition.
- The number and positions of arguments in a function call must match the number and positions of required parameters specified in the function's definition.
- A dictionary associates a set of keys with values. Dictionaries organize data by content rather than position.
- The subscript operator is used to add a new key/value pair to a dictionary or to replace a value associated with an existing key.
- The **dict** type includes methods to access and remove data in a dictionary.
- The **for** loop can traverse the keys of a dictionary. The methods **keys** and **values** return access to a dictionary's keys and values, respectively.
- Bottom-up testing of a program begins by testing its lower-level functions and then testing the functions that depend on those lower-level functions. Top-down testing begins by testing the program's **main** function and then testing the functions on which the **main** function depends. These lower-level functions are initially defined to return their names.

REVIEW QUESTIONS

For questions 1–6, assume that the variable **data** refers to the list **[10, 20, 30]**.

- 1 The expression **data[1]** evaluates to
 - a 10
 - b 20
- 2 The expression **data[1:3]** evaluates to
 - a [10, 20, 30]
 - b [20, 30]

3 The expression `data.index(20)` evaluates to

- a 1
- b 2
- c **True**

4 The expression `data + [40, 50]` evaluates to

- a [10, 60, 80]
- b [10, 20, 30, 40, 50]

5 After the statement `data[1] = 5`, `data` evaluates to

- a [5, 20, 30]
- b [10, 5, 30]

6 After the statement `data.insert(1, 15)`, the original `data` evaluates to

- a [15, 10, 20, 30]
- b [10, 15, 30]
- c [10, 15, 20, 30]

For questions 7–9, assume that the variable `info` refers to the dictionary `{"name": "Sandy", "age": 17}`.

7 The expression `list(info.keys())` evaluates to

- a ("name", "age")
- b ["name", "age"]

8 The expression `info.get("hobbies", None)` evaluates to

- a "knitting"
- b **None**
- c 1000

9 The method to remove an entry from a dictionary is named

- a **delete**
- b **pop**
- c **remove**

10 Which of the following are immutable data structures?

- a dictionaries and lists
- b strings and tuples

PROJECTS

- 1 A group of statisticians at a local college has asked you to create a set of functions that compute the median and mode of a set of numbers, as defined in Section 5.4. Define these functions in a module named **stats.py**. Also include a function named **mean**, which computes the average of a set of numbers. Each function should expect a list of numbers as an argument and return a single number. Each function should return 0 if the list is empty. Include a **main** function that tests the three statistical functions with a given list.
- 2 Write a program that allows the user to navigate the lines of text in a file. The program should prompt the user for a filename and input the lines of text into a list. The program then enters a loop in which it prints the number of lines in the file and prompts the user for a line number. Actual line numbers range from 1 to the number of lines in the file. If the input is 0, the program quits. Otherwise, the program prints the line associated with that number.
- 3 Modify the sentence-generator program of Case Study 5.3 so that it inputs its vocabulary from a set of text files at startup. The filenames are **nouns.txt**, **verbs.txt**, **articles.txt**, and **prepositions.txt**. (*Hint:* Define a single new function, **getWords**. This function should expect a filename as an argument. The function should open an input file with this name, define a temporary list, read words from the file, and add them to the list. The function should then convert the list to a tuple and return this tuple. Call the function with an actual filename to initialize each of the four variables for the vocabulary.)
- 4 Make the following modifications to the original sentence-generator program:
 - a The prepositional phrase is optional. (It can appear with a certain probability.)
 - b A conjunction and a second independent clause are optional: The boy took a drink and the girl played baseball.
 - c An adjective is optional: The girl kicked the red ball with a sore foot.You should add new variables for the sets of adjectives and conjunctions.
- 5 In Chapter 4, we developed an algorithm for converting from binary to decimal. You can generalize this algorithm to work for a representation in any base. Instead of using a power of 2, this time you use a power of

the base. Also, you use digits greater than 9, such as A...F, when they occur. Define a function named **repToDecimal** that expects two arguments, a string and an integer. The second argument should be the base. For example, **repToDecimal("10", 8)** returns 8, whereas **repToDecimal("10", 16)** returns 16. The function should use a lookup table to find the value of any digit. Make sure that this table (it is actually a dictionary) is initialized before the function is defined. For its keys, use the 10 decimal digits (all strings) and the letters A...F (all uppercase). The value stored with each key should be the integer that the digit represents. (The letter 'A' associates with the integer value 10, and so on.) The main loop of the function should convert each digit to uppercase, look up its value in the table, and use this value in the computation. Include a **main** function that tests the conversion function with numbers in several bases.

- 6 Define a function **decimalToRep** that returns the representation of an integer in a given base. The two arguments should be the integer and the base. The function should return a string. It should use a lookup table that associates integers with digits. Include a **main** function that tests the conversion function with numbers in several bases.
- 7 Write a program that inputs a text file. The program should print all of the unique words in the file in alphabetical order.
- 8 A file concordance tracks the unique words in a file and their frequencies. Write a program that displays a concordance for a file. The program should output the unique words and their frequencies in alphabetical order.
- 9 In Case Study 5.5, when the patient addresses the therapist personally, the therapist's reply does not change persons appropriately. To see an example of this problem, test the program with "you are not a helpful therapist." Fix this problem by repairing the dictionary of replacements.
- 10 Conversations often shift focus to earlier topics. Modify the therapist program to support this capability. Add each patient input to a history list. Then, occasionally choose an element at random from this list, change persons, and prepend the qualifier "Earlier you said that" to this reply. Make sure that this option is triggered only after several exchanges have occurred.

[CHAPTER] 6

Design with Functions

After completing this chapter, you will be able to:

- Explain why functions are useful in structuring code in a program
- Employ top-down design to assign tasks to functions
- Define a recursive function
- Explain the use of the namespace in a program and exploit it effectively
- Define a function with required and optional parameters
- Use higher-order functions for mapping, filtering, and reducing

Design is important in many fields. The architect who designs a building, the engineer who designs a bridge or a new automobile, and the politician, advertising executive, or army general who designs the next campaign must organize the structure of a system and coordinate the actors within it to achieve its purpose. Design is equally important in constructing software systems, some of which are the most complex artifacts ever built by human beings. In this chapter, we explore the use of functions to design software systems.

6.1

Functions as Abstraction Mechanisms

Thus far in this book, our programs have consisted of algorithms and data structures, expressed in the Python programming language. The algorithms in turn are composed of built-in operators, control statements, calls to built-in functions, and programmer-defined functions, which were introduced in Chapter 5.

Strictly speaking, functions are not necessary. It is possible to construct any algorithm using only Python's built-in operators and control statements. However, in any significant program, the resulting code would be extremely complex, difficult to verify, and almost impossible to maintain.

The problem is that the human brain can wrap itself around just a few things at once (psychologists say three things comfortably, and at most seven). People cope with complexity by developing a mechanism to simplify or hide it. This mechanism is called an **abstraction**. Put most plainly, an abstraction hides detail and thus allows a person to view many things as just one thing. We use abstractions to refer to the most common tasks in everyday life. For example, consider the expression “doing my laundry.” This expression is simple, but refers to a complex process that involves fetching dirty clothes from the hamper, separating them into whites and colors, loading them into the washer, transferring them to the dryer, and folding them and putting them into the dresser. Indeed, without abstractions, most of our everyday activities would be impossible to discuss, plan, or carry out. Likewise, effective designers must invent useful abstractions to control complexity. In this section, we examine the various ways in which functions serve as abstraction mechanisms in a program.

6.1.1

Functions Eliminate Redundancy

The first way that functions serve as abstraction mechanisms is by eliminating redundant, or repetitious, code. To explore the concept of redundancy, let's look at a function named **sum**, which returns the sum of the numbers within a given range of numbers. Here is the definition of **sum**, followed by a session showing its use:

```
def sum(lower, upper):  
    """  
    Arguments: A lower bound and an upper bound  
    Returns: the sum of the numbers between the arguments  
             and including them  
    """
```

continued

```

result = 0
while lower <= upper:
    result += lower
    lower += 1
return result

>>> sum(1, 4)      # The summation of the numbers 1..4
10
>>> sum(50, 100)   # The summation of the numbers 50..100
3825

```

If the **sum** function didn't exist, the programmer would have to write the entire algorithm every time a summation is computed. In a program that must calculate multiple summations, the same code would appear multiple times. In other words, redundant code would be included in the program. Code redundancy is bad for several reasons. For one thing, it requires the programmer to laboriously enter or copy the same code over and over again, and to get it correct every time. Then, if the programmer decides to improve the algorithm by adding a new feature or making it more efficient, he or she has to revise each instance of the redundant code throughout the entire program. As you can imagine, this would be a maintenance nightmare.

By relying on a single function definition, instead of multiple instances of redundant code, the programmer frees herself to write only a single algorithm in just one place—say, in a library module. Any other module or program can then import the function for its use. Once imported, the function can be called as many times as necessary. When the programmer needs to debug, repair, or improve the function, she needs to edit and test only the single function definition. There is no need to edit the parts of the program that call the function.

6.1.2 Functions Hide Complexity

Another way that functions serve as abstraction mechanisms is by hiding complicated details. To understand why this is true, let's return again to the **sum** function. Although the idea of summing a range of numbers is simple, the code for computing a summation is not. We're not just talking about the amount or length of the code, but also about the number of interacting components. There are three variables to manipulate, as well as count-controlled loop logic to construct.

Now suppose, somewhat unrealistically, that only one summation is performed in a program, and in no other program, ever again. Who needs a function now? Well, it all depends on the complexity of the surrounding code. Remember that the programmers responsible for maintaining a program can wrap their brains around just a few things at a time. If the code for the summation is placed in a context of code that is even slightly complex, the increase in complexity might be enough to result in conceptual overload for the poor programmers.

A function call expresses the idea of a process to the programmer, without forcing him or her to wade through the complex code that realizes that idea. As in other areas of science and engineering, the simplest accounts and descriptions are generally the best.

6.1.3 Functions Support General Methods with Systematic Variations

An algorithm is a **general method** for solving a class of problems. The individual problems that make up a class of problems are known as **problem instances**. The problem instances for our summation algorithm are the pairs of numbers that specify the lower and upper bounds of the range of numbers to be summed. The problem instances of a given algorithm can vary from program to program, or even within different parts of the same program. When you design an algorithm, it should be general enough to provide a solution to many problem instances, not just one or a few of them. In other words, a function should provide a general method with systematic variations.

The **sum** function contains both the code for the summation algorithm and the means of supplying problem instances to this algorithm. The problem instances are the data sent as arguments to the function. The parameters or argument names in the function's header behave like variables waiting to be assigned data whenever the function is called.

If designed properly, a function's code captures an algorithm as a general method for solving a class of problems. The function's arguments provide the means for systematically varying the problem instances that its algorithm solves. Additional arguments can broaden the range of problems that are solvable. For example, the **sum** function could take a third argument that specifies the step to take between numbers in the range. We will examine shortly how to provide additional arguments that do not add complexity to a function's default uses.

6.1.4

Functions Support the Division of Labor

In a well-organized system, whether it is a living thing or something created by humans, each part does its own job or plays its own role in collaborating to achieve a common goal. Specialized tasks get divided up and assigned to specialized agents. Some agents might assume the role of managing the tasks of others or coordinating them in some way. But, regardless of the task, good agents mind their own business and do not try to do the jobs of others.

A poorly organized system, by contrast, suffers from agents performing tasks for which they are not trained or designed, or from agents who are busybodies who do not mind their own business. Division of labor breaks down.

In a computer program, functions can enforce a division of labor. Ideally, each function performs a single coherent task, such as computing a summation or formatting a table of data for output. Each function is responsible for using certain data, computing certain results, and returning these to the parts of the program that requested them. Each of the tasks required by a system can be assigned to a function, including the tasks of managing or coordinating the use of other functions. In the sections that follow, we examine several design strategies that employ functions to enforce a division of labor in programs.

6.1

Exercises

- 1 Anne complains that defining functions to use in her programs is a lot of extra work. She says she can finish her programs much more quickly if she just writes them using the basic operators and control statements. State three reasons why her view is shortsighted.
- 2 Explain how an algorithm solves a general class of problems and how a function definition in particular can support this property of an algorithm.

6.2 Problem Solving with Top-Down Design

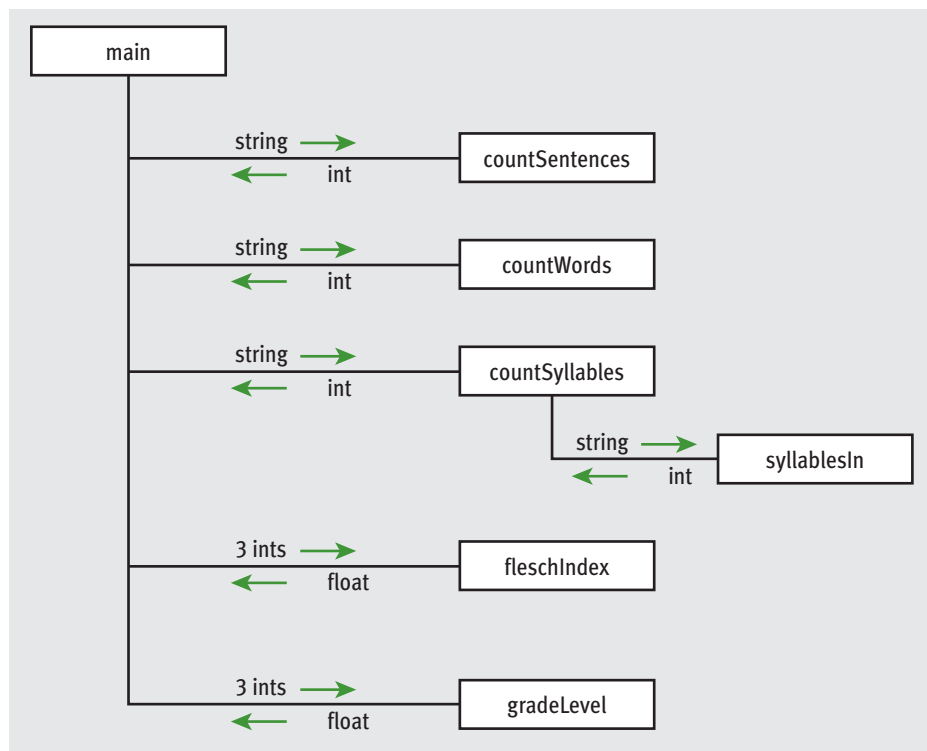
One popular design strategy for programs of any significant size and complexity is called **top-down design**. This strategy starts with a global view of the entire problem and breaks the problem into smaller, more manageable subproblems—a process known as **problem decomposition**. As each subproblem is isolated, its solution is assigned to a function. Problem decomposition may continue down to lower levels, because a subproblem might in turn contain two or more lower-level problems to solve. As functions are developed to solve each subproblem, the solution to the overall problem is gradually filled out in detail. This process is also called **stepwise refinement**.

Our early program examples in Chapters 1–4 were simple enough that they could be decomposed into three parts—the input of data, its processing, and the output of results. None of these parts required more than one or two statements of code, and they all appeared in a single sequence of statements.

However, beginning with the text-analysis program of Chapter 4, our case study problems became complicated enough to warrant decomposition and assignment to additional programmer-defined functions. Because each problem had a different structure, the design of the solution took a slightly different path. This section revisits each program, to explore how their designs took shape.

6.2.1 The Design of the Text-Analysis Program

Although we did not actually structure the text-analysis program (Section 4.7) in terms of programmer-defined functions, we can now explore how that could have been done. The program requires fairly simple input and output components, so these can be expressed as statements within a **main** function. However, the processing of the input is complex enough to decompose into smaller subprocesses, such as obtaining the counts of the sentences, words, and syllables and calculating the readability scores. Generally, you develop a new function for each of these computational tasks. The relationships among the functions in this design are expressed in the structure chart shown in Figure 6.1. A **structure chart** is a diagram that shows the relationships among a program's functions and the passage of data between them.



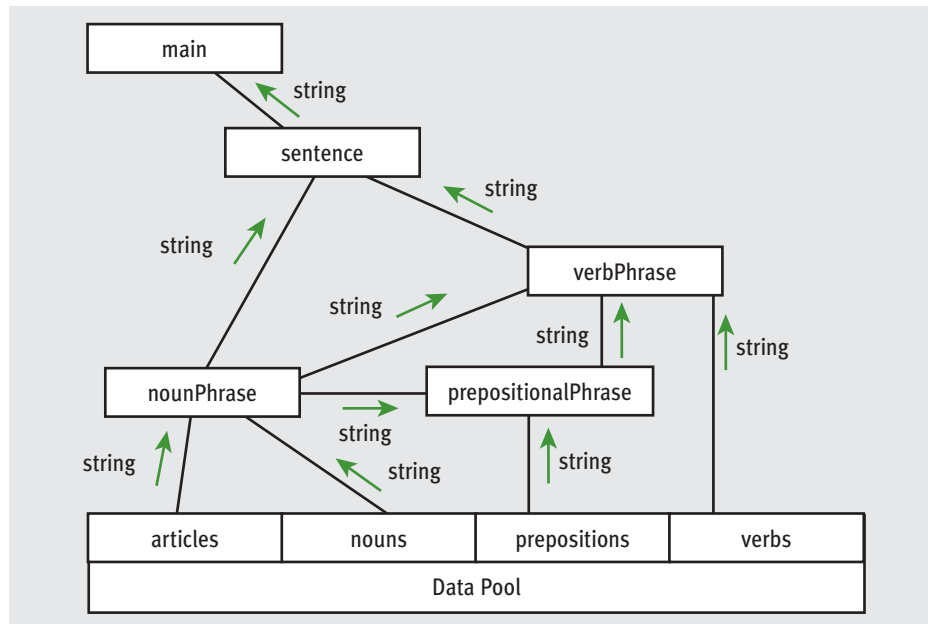
[FIGURE 6.1] A structure chart for the text-analysis program

Each box in the structure chart is labeled with a function name. The **main** function at the top is where the design begins, and decomposition leads us to the lower-level functions on which **main** depends. The lines connecting the boxes are labeled with data type names, and arrows indicate the flow of data between them. For example, the function **countSentences** takes a string as an argument and returns the number of sentences in that string. Note that all functions except one are just one level below **main**. Because this program does not have a deep structure, the programmer can develop it quickly just by thinking of the results that **main** needs to obtain from its collaborators.

6.2.2 The Design of the Sentence-Generator Program

From a global perspective, the sentence-generator program (Section 5.3) consists of a main loop in which sentences are generated a user-specified number of times, until the user enters 0. The I/O and loop logic are simple enough to place in the **main** function. The rest of the design involves generating a sentence.

Here, you decompose the problem by simply following the grammar rules for phrases. To generate a sentence, you generate a noun phrase followed by a verb phrase, and so on. Each of the grammar rules poses a problem that is solved by a single function. The top-down design flows out of the top-down structure of the grammar. The structure chart for the sentence generator is shown in Figure 6.2.



[FIGURE 6.2] A structure chart for the sentence-generator program

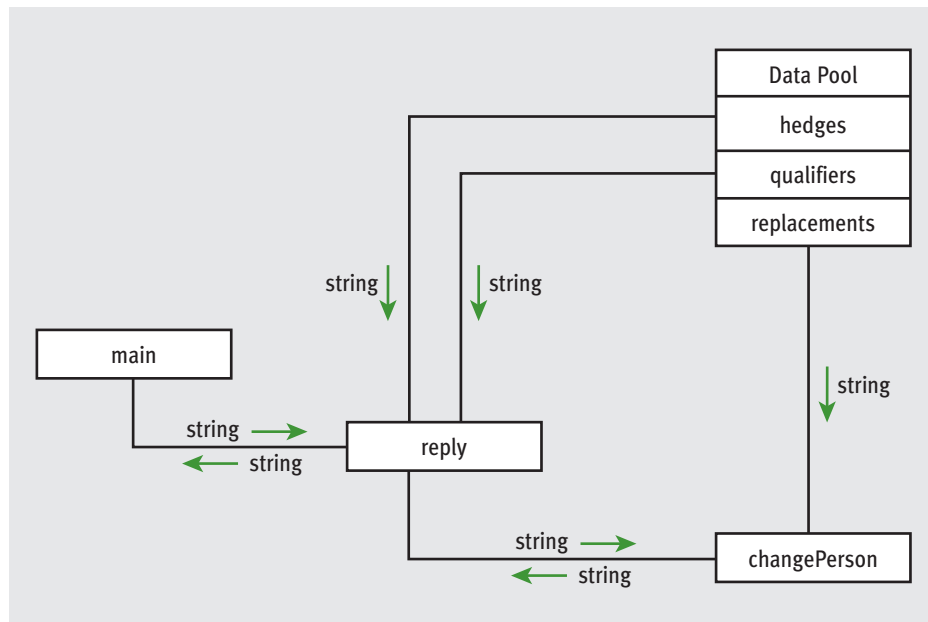
The structure of a problem can often give you a pattern for designing the structure of the program to solve it. In the case of the sentence generator, the structure of the problem comes from the grammar rules, although they are not explicit data structures in the program. In later chapters, we will see many examples of program designs that also mirror the structure of the data being processed.

The design of the sentence generator differs from the design of the text analyzer in one other important way. The functions in the text analyzer all receive data from the **main** function via parameters or arguments. By contrast, the functions in the sentence generator receive their data from a common pool of data defined at the beginning of the module and shown at the bottom of Figure 6.2. This pool of data could equally well have been set up within the **main** function and passed as arguments to each of the other functions. However, this alternative also would require passing arguments to functions that do not actually use them. For

example, **prepositionalPhrase** would have to receive arguments for **articles** and **nouns** as well as **prepositions**, so that it could transmit the first two structures to **nounPhrase**. Using a common pool of data rather than function arguments in this case simplifies the design and makes program maintenance easier.

6.2.3 The Design of the Doctor Program

At the top level, the designs of the doctor program (Section 5.5) and the sentence-generator program are similar. Both programs have main loops that take a single user input and print a result. The structure chart for the doctor program is shown in Figure 6.3.



[FIGURE 6.3] A structure chart for the doctor program

The doctor program actually processes the input by responding to it as an agent would in a conversation. Thus, the responsibility for responding is delegated to the **reply** function. Note that the two functions **main** and **reply** have distinct responsibilities. The job of **main** is to handle user interaction with the program, whereas **reply** is responsible for implementing the “doctor logic” of generating an appropriate reply. The assignment of roles and responsibilities to different actors in a program is also called **responsibility-driven design**. The

division of responsibility between functions that handle user interaction and functions that handle data processing is one that we will see again and again in the coming chapters.

If there were only one way to reply to the user, the problem of how to reply would not be further decomposed. However, because there are at least two options, **reply** is given the task of implementing the logic of choosing one of them, and asks for help from other functions, such as **changePerson**, to carry out each option.

Separating the logic of choosing a task from the process of carrying out a task makes the program more maintainable. To add a new strategy for replying, you add a new choice to the logic of **reply**, and then add the function that carries out this option. If you want to alter the likelihood of a given option, you just modify a line of code in **reply**.

The data flow scheme used in the doctor program combines the strategies used in the text analyzer and the sentence generator. The doctor program's functions receive their data from two sources. The patient's input string is passed as an argument to **reply** and **changePerson**, whereas the qualifiers, hedges, and pronoun replacements are looked up in a common pool of data defined at the beginning of the module. Once again, the use of a common pool of data allows the program to grow easily, as new data sources, such as the history list suggested in Programming Project 5.10, are added to the program.

We conclude this section with an old adage that captures the essence of top-down design. When in doubt about the solution to a problem, pass the buck to someone else. If you choose the right agents, the buck ultimately stops at an agent who has no doubt about how to solve the problem.

6.2 Exercises

- 1 Draw a structure chart for one of the solutions to the programming projects of Chapters 4 and 5. The program should include at least two function definitions other than the **main** function.
- 2 Describe the processes of top-down design and stepwise refinement. Where does the design start, and how does it proceed?

6.3 Design with Recursive Functions

In top-down design, you decompose a complex problem into a set of simpler problems and solve these with different functions. In some cases, you can decompose a complex problem into smaller problems of exactly the same form. In these cases, the subproblems can all be solved by using the same function. This design strategy is called **recursive design**, and the resulting functions are called **recursive functions**.

6.3.1 Defining a Recursive Function

A recursive function is a function that calls itself. To prevent a function from repeating itself indefinitely, it must contain at least one selection statement. This statement examines a condition called a **base case** to determine whether to stop or to continue with another **recursive step**.

Let's examine how to convert an iterative algorithm to a recursive function. Here is a definition of a function **displayRange** that prints the numbers from a lower bound to an upper bound:

```
def displayRange(lower, upper):  
    """Outputs the numbers from lower to upper."""  
    while lower <= upper:  
        print(lower)  
        lower = lower + 1
```

How would we go about converting this function to a recursive one? First, you should note two important facts:

- 1 The loop's body continues execution while **lower** <= **upper**.
- 2 When the function executes, **lower** is incremented by 1, but **upper** never changes.

The equivalent recursive function performs similar primitive operations, but the loop is replaced with a selection statement, and the assignment statement is

replaced with a **recursive call** of the function. Here is the code with these changes:

```
def displayRange(lower, upper):  
    """Outputs the numbers from lower to upper."""  
    if lower <= upper:  
        print(lower)  
        displayRange(lower + 1, upper)
```

Although the syntax and design of the two functions are different, the same algorithmic process is executed. Each call of the recursive function visits the next number in the sequence, just as the loop does in the iterative version of the function.

Most recursive functions expect at least one argument. This data value is used to test for the base case that ends the recursive process, and also is modified in some way before each recursive step. The modification of the data value should produce a new data value that allows the function to reach the base case eventually. In the case of **displayRange**, the value of the argument **lower** is incremented before each recursive call so that it eventually exceeds the value of the argument **upper**.

Our next example is a recursive function that builds and returns a value. Earlier in this chapter, we defined an iterative version of the **sum** function that expects two arguments named **lower** and **upper**. The **sum** function computes and returns the sum of the numbers between these two values. In the recursive version, **sum** returns 0 if **lower** exceeds **upper** (the base case). Otherwise, the function adds **lower** to the **sum** of **lower + 1** and **upper** and returns this result. Here is the code for this function:

```
def sum(lower, upper):  
    """Returns the sum of the numbers from lower to upper."""  
    if lower > upper:  
        return 0  
    else:  
        return lower + sum(lower + 1, upper)
```

The recursive call of **sum** adds up the numbers from **lower + 1** through **upper**. The function then adds **lower** to this result and returns it.

6.3.2 Tracing a Recursive Function

To get a better understanding of how recursion works, it is helpful to trace its calls. Let's do that for the recursive version of the **sum** function. You add an argument for a margin of indentation and **print** statements to trace the two arguments and the value returned on each call. The first statement on each call computes the indentation, which is then used in printing the two arguments. The value computed is also printed with this indentation just before each call returns. Here is the code, followed by a session showing its use:

```
def sum(lower, upper, margin):
    """Returns the sum of the numbers from lower to upper,
    and outputs a trace of the arguments and return values
    on each call."""
    blanks = " " * margin
    print(blanks, lower, upper)
    if lower > upper:
        print(blanks, 0)
        return 0
    else:
        result = lower + sum(lower + 1, upper, margin + 4)
        print(blanks, result)
        return result

>>> sum(1, 4, 0)
1 4
  2 4
    3 4
      4 4
        5 4
          0
        4
      7
    9
  10
10
>>>
```

The displayed pairs of arguments are indented further to the right as the calls of **sum** proceed. Note that the value of **lower** increases by 1 on each call, whereas the value of **upper** stays the same. The final call of **sum** returns 0. As the recursion unwinds, each value returned is aligned with the arguments above it and increases by the current value of **lower**. This type of tracing can be a useful debugging tool for recursive functions.

6.3.3 Using Recursive Definitions to Construct Recursive Functions

Recursive functions are frequently used to design algorithms for computing values that have a **recursive definition**. A recursive definition consists of equations that state what a value is for one or more base cases and one or more recursive cases. For example, the Fibonacci sequence is a series of values with a recursive definition. The first and second numbers in the Fibonacci sequence are 1. Thereafter, each number in the sequence is the sum of its two predecessors, as follows:

```
1 1 2 3 5 8 13 . . .
```

More formally, a recursive definition of the n th Fibonacci number is the following:

```
Fib(n) = 1, when n = 1 or n = 2  
Fib(n) = Fib(n - 1) + Fib(n - 2), for all n > 2
```

Given this definition, you can construct a recursive function that computes and returns the n th Fibonacci number. Here it is:

```
def fib(n):  
    """Returns the nth Fibonacci number."""  
    if n < 3:  
        return 1  
    else:  
        return fib(n - 1) + fib(n - 2)
```

Note that the base case as well as the two recursive steps return values to the caller.

6.3.4 Recursion in Sentence Structure

Recursive solutions can often flow from the structure of a problem. For example, the structure of sentences in a language can be highly recursive. A noun phrase (such as “the ball”) can be modified by a prepositional phrase (such as “on the

bench”), which also contains another noun phrase. If you use this modified version of the noun phrase rule in the sentence generator (Section 5.3), the **nounPhrase** function would call the **prepositionalPhrase** function, which in turn calls **nounPhrase** again. This phenomenon is known as **indirect recursion**. To keep this process from going on forever, **nounPhrase** must also have the option to not generate a prepositional phrase. Here is a statement of the modified rule, which expresses an optional phrase within the square brackets:

```
Noun phrase = Article Noun [Prepositional phrase]
```

The code for a revised **nounPhrase** function generates a modifying prepositional phrase approximately 25% of the time:

```
def nounPhrase():
    """Returns a noun phrase, which is an article followed
    by a noun and an optional prepositional phrase."""
    phrase = random.choice(articles) + " " + random.choice(nouns)
    prob = random.randint(1, 4)
    if prob == 1:
        return phrase + " " + prepositionalPhrase()
    else:
        return phrase
```

You can use a similar strategy to generate sentences that consist of two or more independent clauses connected by conjunctions, such as “One programmer uses recursion and another programmer uses loops.”

6.3.5 Infinite Recursion

Recursive functions tend to be simpler than the corresponding loops, but still require thorough testing. One design error that might trip up a programmer occurs when the function can (theoretically) continue executing forever, a situation known as **infinite recursion**. Infinite recursion arises when the programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process. In fact, the Python virtual machine eventually

runs out of memory resources to manage the process, so it halts execution with an error message. The next session defines a function that leads to this result:

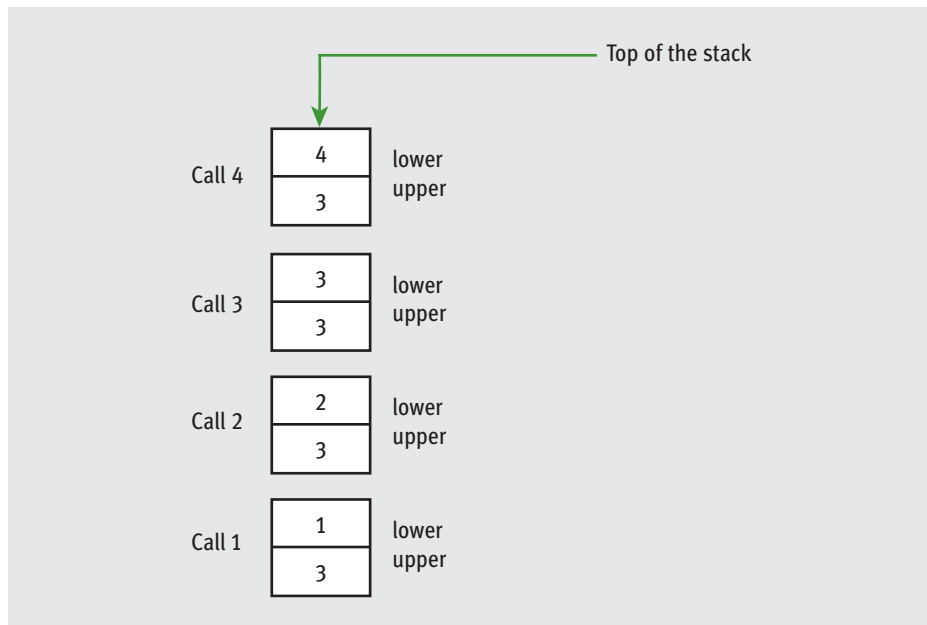
```
>>> def runForever(n):
    if n > 0:
        runForever(n)
    else:
        runForever(n - 1)

>>> runForever(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 3, in runForever
RuntimeError: maximum recursion depth exceeded
>>>
```

The Python virtual machine keeps calling **runForever(1)** until there is no memory left to support another recursive call. Unlike an infinite loop, an infinite recursion eventually halts execution with an error message.

6.3.6 The Costs and Benefits of Recursion

Although recursive solutions are often more natural and elegant than their iterative counterparts, they come with a cost. The run-time system on a real computer, such as the Python virtual machine, must devote some overhead to recursive function calls. At program startup, the PVM reserves an area of memory named a **call stack**. For each call of a function, the PVM must allocate on the call stack a small chunk of memory called a **stack frame**. In this type of storage, the system places the values of the arguments and the return address for the particular function call. Space for the function call's return value is also reserved in its stack frame. When a call returns or completes its execution, the return address is used to locate the next instruction in the caller's code, and the memory for the stack frame is deallocated. The stack frames for the process generated by **displayRange(1, 3)** are shown in Figure 6.4. The frames in the figure include storage for the function's arguments only.



[FIGURE 6.4] The stack frames for `displayRange(1, 3)`

Although this sounds like a complex process, the PVM handles it easily. However, when a function invokes hundreds or even thousands of recursive calls, the amount of extra resources required, both in processing time and in memory usage, can add up to a significant performance hit. When, because of a design error, the recursion is infinite, the stack frames are added until the PVM runs out of memory, which halts the program with an error message.

By contrast, the same problem can often be solved using a loop with a constant amount of memory, in the form of two or three variables. Because the amount of memory needed for the loop does not grow with the size of the problem's data set, the amount of processing time for managing this memory does not grow, either.

Despite these words of caution, we encourage you to consider developing recursive solutions when they seem natural, particularly when the problems themselves have a recursive structure. Testing can reveal performance bottlenecks that might lead you to change the design to an iterative one. Smart compilers also exist that can optimize some recursive functions by translating them to iterative machine code. Finally, as we will see later in this book, some problems with an iterative solution must still use an explicit stack-like data structure, so a recursive solution might be simpler and no less efficient.

Recursion is a very powerful design technique that is used throughout computer science. We will return to it in later chapters.

6.3 Exercises

- 1 In what way is a recursive design different from top-down design?
- 2 The factorial of a positive integer **n**, **fact(n)**, is defined recursively as follows:

```
fact(n) = 1, when n = 1
fact(n) = n * fact(n - 1), otherwise
```

Define a recursive function **fact** that returns the factorial of a given positive integer.

- 3 Describe the costs and benefits of defining and using a recursive function.
- 4 Explain what happens when the following recursive function is called with the value 4 as an argument:

```
def example(n):
    if n > 0:
        print(n)
        example(n - 1)
```

- 5 Explain what happens when the following recursive function is called with the value 4 as an argument:

```
def example(n):
    if n > 0:
        print(n)
        example(n)
    else:
        example(n - 1)
```

- 6 Explain what happens when the following recursive function is called with the values **"hello"** and **0** as arguments:

```
def example(aString, index):
    if index < len(aString):
        example(aString, index + 1)
        print(aString[index], end="")
```

- 7 Explain what happens when the following recursive function is called with the values **"hello"** and **0** as arguments:

```
def example(aString, index):
    if index == len(aString):
        return ""
    else:
        return aString[index] + example(aString, index + 1)
```

6.4

Case Study: Gathering Information from a File System

Modern file systems come with a graphical browser, such as Microsoft's Windows Explorer or Apple's Finder. These browsers allow the user to navigate directories by selecting icons of folders, opening these by double-clicking, and selecting commands from a drop-down menu. Information on a directory or a file, such as the size and contents, is also easily obtained in several ways.

Users of terminal-based user interfaces must rely on entering the appropriate commands at the terminal prompt to perform all of these functions. In this case study, we develop a simple terminal-based file system navigator that provides some information about the system. In the process, we will have an opportunity to exercise some skills in top-down design and recursive design.

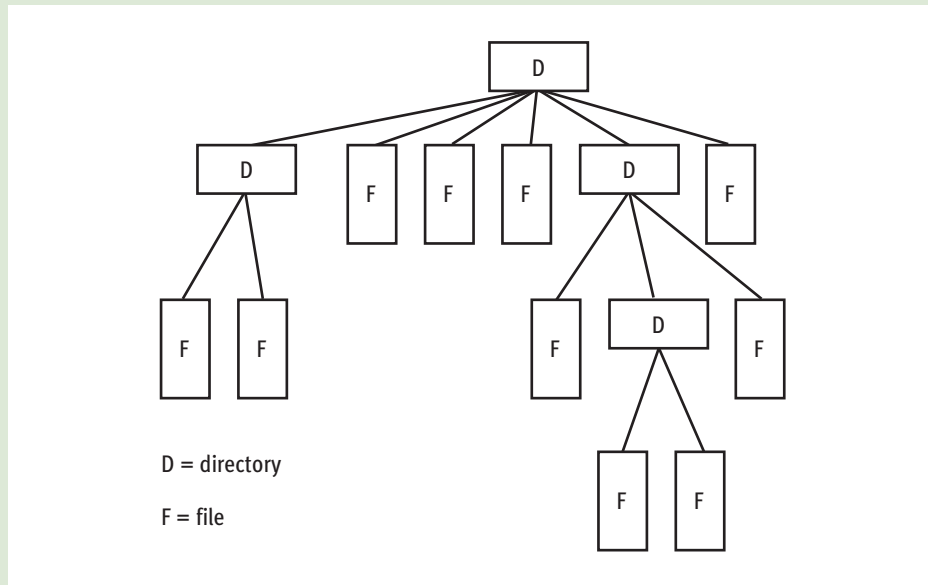
6.4.1

Request

Write a program that allows the user to obtain information about the file system.

6.4.2 Analysis

File systems are tree-like structures, as shown in Figure 6.5.



[FIGURE 6.5] The structure of a file system

At the top of the tree is the **root directory**. Under the root are files and subdirectories. Each directory in the system except the root lies within another directory called its **parent**. For example, in Figure 6.5, the root directory contains four files and two subdirectories. On a UNIX-based file system, the **path** to a given file or directory in the system is a string that starts with the / (forward slash) symbol (the root), followed by the names of the directories traversed to reach the file or directory. The / (forward slash) symbol also separates each name in the path. Thus, the path to the file for this chapter on Ken's laptop might be the following:

```
/Users/KenLaptop/Book/Chapter6/Chapter6.doc
```

On a Windows-based file system, the \ symbol is used instead of the / symbol.

The program we will design in this case study is named **filesys.py**. It provides some basic browsing capability, as well as options that allow you to search for a given filename and find statistics on the number of files and their size in a directory.

At program startup, the current working directory (CWD) is the directory containing the Python program file. The program should display the path of the CWD, a menu of command options, and a prompt for a command, as shown in Figure 6.6.

```

/Users/KenLaptop/Book/Chapter6
1  List the current directory
2  Move up
3  Move down
4  Number of files in the directory
5  Size of the directory in bytes
6  Search for a filename
7  Quit the program
Enter a number:
```

[FIGURE 6.6] The command menu of the **filesys** program

When the user enters a command number, the program runs the command, which may display further information, and the program displays the CWD and command menu again. An unrecognized command produces an error message, and command number 7 quits the program. Table 6.1 summarizes what the commands do.

COMMAND	WHAT IT DOES
List the current working directory	Prints the names of the files and directories in the current working directory (CWD).
Move up	If the CWD is not the root, move to the parent directory and make it the CWD.
Move down	Prompts the user for a directory name. If the name is not in the CWD, print an error message; otherwise, move to this directory and make it the CWD.
Number of files in the directory	Prints the number of files in the CWD and all of its subdirectories.
Size of the directory in bytes	Prints the total number of bytes used by the files in the CWD and all of its subdirectories.
Search for a filename	Prompts the user for a search string. Prints a list of all the filenames (with their paths) that contain the search string, or “String not found.”
Quit the program	Prints a signoff message and exits the program.

[TABLE 6.1] The commands in the **filesys** program

6.4.3 Design

You can structure the program according to two sets of tasks: those concerned with implementing a menu-driven command processor, and those concerned with executing the commands. The first group of operations includes the **main** function. In the following discussion, we work top-down and begin by examining the first group of operations.

As in many of the programs we have examined recently in this book, the **main** function contains a driver loop. This loop prints the CWD and the menu, calls other functions to input and run the commands, and breaks with a signoff message when the command is to quit. Here is the pseudocode:

```
function main()
    while True
        print(os.getcwd())
        print(MENU)
        Set command to acceptCommand()
        runCommand(command)
        if command == QUIT
            print("Have a nice day!")
            break
```

Note that **MENU** and **QUIT** are variables initialized to the appropriate strings before **main** is defined. The **acceptCommand** function loops until the user enters a number in the range of the valid commands. These commands are specified in a tuple named **COMMANDS** that is also initialized before the function is defined. The function thus always returns a valid command number.

The **runCommand** function expects a valid command number as an argument. The function uses a multi-way selection statement to select and run the operation corresponding to the command number. When the result of an operation is returned, it is printed with the appropriate labeling.

That's it for the menu-driven command processor. Although there are other possible approaches, this design makes it possible to add new commands to the program fairly easily.

The operations required to list the contents of the CWD, move up, and move down are fairly simple and need no real design work. They involve the use of functions in the **os** and **os.path** modules to list the directory, change it, and test a string to see if it is the name of a directory. The implementation shows the details.

The other three operations all involve traversals of the directory structure in the CWD. During these traversals, every file and every subdirectory are visited. Directory structure is in fact recursive: each directory can contain files (base cases) and other directories (recursive steps). Thus, we can develop a recursive design for each operation.

The **countFiles** function expects the path of a directory as an argument and returns the number of files in this directory and all of its subdirectories. If there are no subdirectories in the argument directory, the function just counts the files and returns this value. If there is a subdirectory, the function moves down to it, counts the files (recursively) in it, adds the result to its total, and then moves back up to the parent directory. Here is the pseudocode:

```
function countFiles(path)
    Set count to 0
    Set lyst to os.listdir(path)
    for element in lyst
        if os.path.isfile(element)
            count += 1
        else:
            os.chdir(element)
            count += countFiles(os.getcwd())
            os.chdir("../")
    return count
```

The **countBytes** function expects a path as an argument and returns the total number of bytes in that directory and all of its subdirectories. Its design is quite similar to **countFiles**.

The **findFiles** function accumulates a list of the filenames, including their paths, that contain a given target string, and returns this list. Its structure is similar to the other two recursive functions, but the **findFiles** function builds a list rather than a number. When the function encounters a target file, its name is appended to the path, and then the result string is appended to the list of files. We use the module variable **os.sep** to obtain the appropriate slash symbol (/ or \) on the current file system. When the function encounters a directory, it moves to that directory, calls itself with the new CWD, and extends the files list with the resulting list. Here is the pseudocode:

```
function findFiles(target, path)
    files = []
    lyst = os.listdir(path)
```

continued

```

for element in lyst
    if os.path.isfile(element):
        if target in element:
            files.append(path + os.sep + element)
    else:
        os.chdir(element)
        files.extend(findFiles(target, os.getcwd()))
        os.chdir("../")
return files

```

The trick with recursive design is to spot elements in a structure that can be treated as base cases (such as files) and other elements that can be treated as recursive steps (such as directories). The recursive algorithms for processing these structures flow naturally from these insights.

6.4.4 Implementation (Coding)

Near the beginning of the program code, we find the important variables, with the functions listed in a top-down order.

```

"""
Program: filesystem.py
Author: Ken

Provides a menu-driven tool for navigating a file system
and gathering information on files.
"""

import os, os.path

QUIT = '7'

COMMANDS = ('1', '2', '3', '4', '5', '6', '7')

MENU = """1  List the current directory
2  Move up
3  Move down
4  Number of files in the directory
5  Size of the directory in bytes
6  Search for a filename
7  Quit the program"""

def main():
    while True:

```

continued

```

        print(os.getcwd())
        print(MENU)
        command = acceptCommand()
        runCommand(command)
        if command == QUIT:
            print("Have a nice day!")
            break

def acceptCommand():
    """Inputs and returns a legitimate command number."""
    while True:
        command = input("Enter a number: ")
        if not command in COMMANDS:
            print("Error: command not recognized")
        else:
            return command

def runCommand(command):
    """Selects and runs a command."""
    if command == '1':
        listCurrentDir(os.getcwd())
    elif command == '2':
        moveUp()
    elif command == '3':
        moveDown(os.getcwd())
    elif command == '4':
        print("The total number of files is", \
              countFiles(os.getcwd()))
    elif command == '5':
        print("The total number of bytes is", \
              countBytes(os.getcwd()))
    elif command == '6':
        target = input("Enter the search string: ")
        fileList = findFiles(target, os.getcwd())
        if not fileList:
            print("String not found")
        else:
            for f in fileList:
                print(f)

def listCurrentDir(dirName):
    """Prints a list of the cwd's contents."""
    lyst = os.listdir(dirName)
    for element in lyst: print(element)

def moveUp():
    """Moves up to the parent directory."""
    os.chdir("..")

```

continued

```

def moveDown(currentDir):
    """Moves down to the named subdirectory if it exists."""
    newDir = input("Enter the directory name: ")
    if os.path.exists(currentDir + os.sep + newDir) and \
        os.path.isdir(newDir):
        os.chdir(newDir)
    else:
        print("ERROR: no such name")

def countFiles(path):
    """Returns the number of files in the cwd and
    all its subdirectories."""
    count = 0
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            count += 1
        else:
            os.chdir(element)
            count += countFiles(os.getcwd())
            os.chdir("../")
    return count

def countBytes(path):
    """Returns the number of bytes in the cwd and
    all its subdirectories."""
    count = 0
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            count += os.path.getsize(element)
        else:
            os.chdir(element)
            count += countBytes(os.getcwd())
            os.chdir("../")
    return count

def findFiles(target, path):
    """Returns a list of the filenames that contain
    the target string in the cwd and all its subdirectories."""
    files = []
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            if target in element:
                files.append(path + os.sep + element)

```

continued

```

        else:
            os.chdir(element)
            files.extend(findFiles(target, os.getcwd()))
            os.chdir("..")
    return files

main()

```

6.5 Managing a Program's Namespace

Throughout this book, we have tried to behave like good authors by choosing our words (the code used in our programs) carefully. We have taken care to select variable names that reflect their purpose in a program or the character of the objects in a given problem domain. Of course, these variable names are meaningful only to us, the human programmers. To the computer, the only “meaning” of a variable name is the value to which it happens to refer at any given point in program execution. The computer can keep track of these values easily. However, a programmer charged with editing and maintaining code can occasionally get lost as a program gets larger and more complex. In this section, you learn more about how a program’s **namespace**—that is, the set of its variables and their values—is structured and how you can control it via good design principles.

6.5.1 Module Variables, Parameters, and Temporary Variables

We begin by analyzing the namespace of the doctor program of Case Study 5.5. This program includes many variable names; for the purposes of this example, we will focus on the code for the variable **replacements** and the function **changePerson**.

```

replacements = {"I":"you", "me":"you", "my":"your",
               "we":"you", "us":"you", "mine":"yours"}

def changePerson(sentence):
    """Replaces first person pronouns with second person
    pronouns."""
    words = sentence.split()
    replyWords = []
    for word in words:
        replyWords.append(replacements.get(word, word))
    return " ".join(replyWords)

```

This code appears in the file **doctor.py**, so its module name is **doctor**. The names in this code fall into four categories, depending on where they are introduced:

- 1 **Module variables.** The names **replacements** and **changePerson** are introduced at the level of the module. Although **replacements** names a dictionary and **changePerson** names a function, they are both considered variables. You can see the module variables of the **doctor** module by importing it and entering **dir(doctor)** at a shell prompt. When module variables are introduced in a program, they are immediately given a value.
- 2 **Parameters.** The name **sentence** is a parameter of the function **changePerson**. A parameter name behaves like a variable and is introduced in a function or method header. The parameter does not receive a value until the function is called.
- 3 **Temporary variables.** The names **words**, **replyWords**, and **word** are introduced in the body of the function **changePerson**. Like module variables, temporary variables receive their values as soon as they are introduced.
- 4 **Method names.** The names **split** and **join** are introduced or defined in the **str** type. As mentioned earlier, a method reference always uses an object, in this case, a string, followed by a dot and the method name.

Our first simple programs contained module variables only. The use of function definitions brought parameters and temporary variables into play. We now explore the significance of these distinctions.

6.5.2 Scope

In ordinary writing, the meaning of a word often depends on its surrounding context. For example, in the sports section of the newspaper, the word “bat” means a stick for hitting baseballs, whereas in a story about vampires it means a flying mammal. In a program, the context that gives a name a meaning is called its **scope**. In Python, a name’s scope is the area of program text in which the name refers to a given value.

Let’s return to our example from the doctor program to determine the scope of each variable. For reasons that will become clear in a moment, it will be easiest if we work outward, starting with temporary variables first.

The scope of the temporary variables **words**, **replyWords**, and **word** is the area of code in the body of the function **changePerson**, just below where each

variable is introduced. In general, the meanings of temporary variables are restricted to the body of the functions in which they are introduced, and are invisible elsewhere in a module. The restricted visibility of temporary variables befits their role as temporary working storage for a function.

The scope of the parameter **sentence** is the entire body of the function **changePerson**. Like temporary variables, parameters are invisible outside the function definitions where they are introduced.

The scope of the module variables **replacements** and **changePerson** includes the entire module below the point where the variables are introduced. This includes the code nested in the body of the function **changePerson**. The scope of these variables also includes the nested bodies of other function definitions that occur *earlier*. This allows these variables to be referenced by any functions, regardless of where they are defined in the module. For example, the **reply** function, which calls **changePerson**, might be defined before **changePerson** in the doctor module.

Although a Python function can reference a module variable for its value, it cannot under normal circumstances assign a new value to a module variable. When such an attempt is made, the PVM creates a new, temporary variable of the same name within the function. The following script shows how this works:

```
x = 5

def f():
    x = 10          # Attempt to reset x

f()                # Does the top-level x change?
print(x)           # No, this displays 5
```

When the function **f** is called, it does not assign 10 to the module variable **x**; instead, it assigns 10 to a temporary variable **x**. In fact, once the temporary variable is introduced, the module variable is no longer visible within function **f**. In any case, the module variable's value remains unchanged by the call. There is a way to allow a function to modify a module variable, but in Chapter 8, we explore a better way to manage common pools of data that require changes.

6.5.3 Lifetime

A computer program has two natures. On the one hand, a program is a piece of text containing names that a human being can read for a meaning. Viewed from this perspective, variables in a program have a scope that determines their visibility.

On the other hand, a program describes a process that exists for a period of time on a real computer. Viewed from this other perspective, a program's variables have another important property called a **lifetime**. A variable's lifetime is the period of time during program execution when the variable has memory storage associated with it. When a variable comes into existence, storage is allocated for it; when it goes out of existence, storage is reclaimed by the PVM.

Module variables come into existence when they are introduced via assignment and generally exist for the lifetime of the program that introduces or imports those module variables. Parameters and temporary variables come into existence when they are bound to values during a function call, but go out of existence when the function call terminates.

The concept of lifetime explains the existence of two variables called **x** in our last example session. The module variable **x** comes into existence before the temporary variable **x** and survives the call of function **f**. During the call of **f**, storage exists for both variables, so their values remain distinct. A similar mechanism for managing the storage associated with the parameters of recursive function calls was discussed in the previous section.

6.5.4 Default (Keyword) Arguments

A function's arguments are one of its most important features. Arguments provide the function's caller with the means of transmitting information to the function. Adding an argument or two to a function can increase its generality by extending the range of situations in which the function can be used. However, programmers often use a function in a restricted set of "essential" situations, in which the extra arguments might be an annoyance. In these cases, the use of the extra arguments should be optional for the caller of the function. When the function is called without the extra arguments, it provides reasonable default values for those arguments that produce the expected results.

For example, Python's **range** function can be called with one, two, or three arguments. When all three arguments are supplied, they indicate a lower bound, an upper bound, and a step value. When only two arguments are given, the step value defaults to 1. When a single argument is given, the step is assumed to be 1, and the lower bound automatically is 0.

The programmer can also specify optional arguments with default values in any function definition. Here is the syntax:

```
def <function name>(<required args>,  
    <key-1> = <val-1>, ... <key-n> = <val-n>)
```


The required arguments are listed first in the function header. These are the ones that are “essential” for the use of the function by any caller. Following the required arguments are one or more **default** or **keyword arguments**. These are assignments of values to the argument names. When the function is called without these arguments, their default values are automatically assigned to them. When the function is called with these arguments, the default values are overridden by the caller’s values.

For example, suppose we define a function, **repToInt**, to convert string representations of numbers in a given base to their integer values (see Chapter 4). The function expects a string representation of the number and an integer base as arguments. Here is the code:

```
def repToInt(repString, base):
    """Converts the repString to an int in the base
    and returns this int."""
    decimal = 0
    exponent = len(repString) - 1
    for digit in repString:
        decimal = decimal + int(digit) * base ** exponent
        exponent -= 1
    return decimal
```

As written, this function can be used to convert string representations in bases 2 through 10 to integers. But suppose that 75% of the time, programmers use the **repToInt** function to convert binary numbers to decimal form. If we alter the function header to provide a default of 2 for **base**, those programmers will be very grateful. Here is the proposed change, followed by a session that shows its impact:

```
def repToInt(repString, base = 2):

>>> repToInt("10", 10)
10
>>> repToInt("10", 8)    # Override the default to here
8
>>> repToInt("10", 2)    # Same as the default, not necessary
2
>>> repToInt("10")       # Base 2 by default
2
>>>
```

When using functions that have default arguments, you must provide the required arguments and place them in the same positions as they are in the function definition's header. The default arguments that follow can be supplied in two ways:

- 1 **By position.** In this case, the values are supplied in the order in which the arguments occur in the function header. Defaults are used for any arguments that are omitted.
- 2 **By keyword.** In this case, one or more values can be supplied in any order, using the syntax **<key> = <value>** in the function call.

Here is an example of a function with one required argument and two default arguments and a session that shows these options:

```
def example(required, option1 = 2, option2 = 3):
    print(required, option1, option2)

>>> example(1)                # Use all the defaults
1 2 3
>>> example(1, 10)            # Override the first default
1 10 3
>>> example(1, 10, 20)        # Override all the defaults
1 10 20
>>> example(1, option2 = 20)   # Override the second default
1 2 20
>>> example(1, option2 = 20, option1 = 10) # Note the order
1 10 20
>>>
```

Default arguments are a powerful way to simplify design and make functions more general.

6.5 Exercises

- 1 Where are module variables, parameters, and temporary variables introduced and initialized in a program?
- 2 What is the scope of a variable? Give an example.
- 3 What is the lifetime of a variable? Give an example.

6.6 Higher-Order Functions (Advanced Topic)

Like any skill, a designer's knack for spotting the need for a function is developed with practice. As you gain experience in writing programs, you will learn to spot common and redundant patterns in the code. One pattern that occurs again and again is the application of a function to a set of values to produce some results. Here are some examples:

- All of the numbers in a text file must be converted to integers or floats after they are input.
- All of the first-person pronouns in a list of words must be changed to the corresponding second-person pronouns in the **doctor** program.
- Only scores above the average are kept in a list of grades.
- The sum of the squares of a list of numbers is computed.

In this section, we learn how to capture these patterns in a new abstraction called a **higher-order function**. For these patterns, a higher-order function expects a function and a set of data values as arguments. The argument function is applied to each data value, and a set of results or a single data value is returned. A higher-order function separates the task of transforming each data value from the logic of accumulating the results.

6.6.1 Functions as First-Class Data Objects

In Python, functions can be treated as **first-class data objects**. This means that they can be assigned to variables (as they are when they are defined), passed as arguments to other functions, returned as the values of other functions, and stored in data structures such as lists and dictionaries. The next session shows some of the simpler possibilities:

```
>>> abs                                # See what a function looks like
<built-in function abs>
>>> import math
>>> math.sqrt
<built-in function sqrt>
>>> f = abs                             # f is an alias for abs
>>> f                                   # Evaluate f
<built-in function abs>
>>> f(-4)                              # Apply f to an argument
4
```

continued

```
>>> funcs = [abs, math.sqrt]      # Put the functions in a list
>>> funcs
[<built-in function abs>, <built-in function sqrt>]
>>> funcs[1](2)                   # Apply math.sqrt to 2
1.4142135623730951
>>>
```

Passing a function as an argument to another function is no different from passing any other datum. The function argument is first evaluated, producing the function itself, and then the parameter name is bound to this value. The function can then be applied to its own argument with the usual syntax. Here is an example, which simply returns the result of an application of any single-argument function to a datum:

```
>>> def example(functionArg, dataArg):
    return functionArg(dataArg)

>>> example(abs, -4)
4
>>> example(math.sqrt, 2)
1.4142135623730951
>>>
```

6.6.2 Mapping

The first type of useful higher-order function to consider is called a **mapping**. This process applies a function to each value in a sequence (such as a list, a tuple, or a string) and returns a new sequence of the results. Python includes a **map** function for this purpose. Suppose we have a list named **words** that contains strings that represent integers. We want to replace each string with the

corresponding integer value. The **map** function easily accomplishes this, as the next session shows:

```
>>> words = ["231", "20", "-45", "99"]
>>> map(int, words)           # Convert all strings to ints
<map object at 0x14cbd90>
>>> words                     # Original list is not changed
['231', '20', '-45', '99']
>>> words = list(map(int, words)) # Reset variable to change it
>>> words
[231, 20, -45, 99]
>>>
```

Note that **map** builds and returns a new map object, which we feed to the **list** function to view the results. We could have written a **for** loop that does the same thing, but that would entail several lines of code instead of the single line of code required for the **map** function. Another reason to use the **map** function is that, in programs that use lists, we might need to perform this task many times; relying on a **for** loop for each instance would entail multiple sections of redundant code. Moreover, the conversion to a list is only necessary for viewing the results; a map object can be passed directly to another **map** function to perform further transformations of the data.

Another good example of a mapping pattern is in the **changePerson** function of the **doctor** program. This function builds a new list of words with the pronouns replaced.

```
def changePerson(sentence):
    """Replaces first person pronouns with second person
    pronouns."""
    words = sentence.split()
    replyWords = []
    for word in words:
        replyWords.append(replacements.get(word, word))
    return " ".join(replyWords)
```

We can simplify the logic by defining an auxiliary function that is then mapped onto the list of words, as follows:

```
def changePerson(sentence):  
    """Replaces first person pronouns with second person  
    pronouns."""  
  
    def getWord(word):  
        replacements.get(word, word)  
  
    return " ".join(map(getWord, sentence.split()))
```

Note that the definition of the function **getWord** is nested within the function **changePerson**. Furthermore, the map object is passed directly to the string method **join** without converting it to a list.

As you can see, the **map** function is extremely useful; any time we can eliminate a loop from a program, it's a win.

6.6.3 Filtering

A second type of higher-order function is called a **filtering**. In this process, a function called a **predicate** is applied to each value in a list. If the predicate returns **True**, the value passes the test and is added to a filter object (similar to a map object). Otherwise, the value is dropped from consideration. The process is a bit like pouring hot water into a filter basket with coffee. The good stuff to drink comes into the cup with the water, and the coffee grounds left behind can be thrown on the garden.

Python includes a **filter** function that is used in the next example to produce a list of the odd numbers in another list:

```
>>> def odd(n): return n % 2 == 1  
  
>>> list(filter(odd, range(10)))  
[1, 3, 5, 7, 9]  
>>>
```

As with the function **map**, the result of the function **filter** can be passed directly to another call of **filter** or **map**. List processing often consists of several mappings and filterings of data, which can be expressed as a series of nested function calls.

6.6.4 Reducing

Our final example of a higher-order function is called a **reducing**. Here we take a list of values and repeatedly apply a function to accumulate a single data value. A summation is a good example of this process. The first value is added to the second value, then the sum is added to the third value, and so on, until the sum of all the values is produced.

The Python **functools** module includes a **reduce** function that expects a function of two arguments and a list of values. The **reduce** function returns the result of applying the function as just described. The following example shows **reduce** used twice—once to produce a sum and once to produce a product:

```
>>> from functools import reduce
>>> def add(x, y): return x + y

>>> def multiply(x, y): return x * y

>>> data = [1, 2, 3, 4]
>>> reduce(add, data)
10
>>> reduce(multiply, data)
24
>>>
```

6.6.5 Using lambda to Create Anonymous Functions

Although the use of higher-order functions can really simplify code, it is somewhat onerous to have to define new functions to supply as arguments to the higher-order functions. For example, the functions **add** and **multiply** will never be used anywhere else in a program, because the operators **+** and ***** are already available. It would be convenient if we could define a function “on the fly,” right at the point of the call of a higher-order function, especially if it is not needed anywhere else.

Python includes a mechanism called **lambda** that allows the programmer to create functions in this manner. A **lambda** is an **anonymous function**. It has no name of its own, but contains the names of its arguments as well as a single expression. When the **lambda** is applied to its arguments, its expression is evaluated, and its value is returned.

The syntax of a **lambda** is very tight and restrictive:

```
lambda <argname-1, ..., argname-n>: <expression>
```

All of the code must appear on one line and, although it is sad, a **lambda** cannot include a selection statement, because selection statements are not expressions. Nonetheless, **lambda** has its virtues. We can now specify addition or multiplication on the fly, as the next session illustrates:

```
>>> data = [1, 2, 3, 4]
>>> reduce(lambda x, y: x + y, data)      # Produce the sum
10
>>> reduce(lambda x, y: x * y, data)      # Produce the product
24
>>>
```

The next example shows the use of **range**, **reduce**, and **lambda** to simplify the definition of the **sum** function discussed earlier in this chapter:

```
def sum(lower, upper):
    """Returns the sum of the numbers from lower to upper."""
    if lower > upper:
        return 0
    else:
        return reduce(lambda x, y: x + y,
                       range(lower, upper + 1))
```

6.6.6 Creating Jump Tables

This chapter's case study contains a menu-driven command processor. When the user selects a command from a menu, the program compares this number to each number in a set of numbers, until a match is found. A function corresponding to this number is then called to carry out the command. The function **runCommand** implemented this process with a long, multi-way selection statement. With more than three options, such statements become tedious to read and hard to maintain. Adding or removing an option also becomes tricky and error-prone.

A simpler way to design a command processor is to use a data structure called a **jump table**. A jump table is a dictionary of functions keyed by command names. At program startup, the functions are defined and then the jump table is loaded with the command names and their associated functions. The function **runCommand** uses its **command** argument to look up the function in the jump table and then calls this function. Here is the modified version of **runCommand**:

```
def runCommand(command):          # How simple can it get?
    jumpTable[command]()
```

Note that this function makes two important simplifying assumptions: the command string is a key in the jump table, and its associated function expects no arguments.

Let's assume that the functions **insert**, **replace**, and **remove** are keyed to the commands **'1'**, **'2'**, and **'3'**, respectively. Then the setup of the jump table is straightforward:

```
# The functions named insert, replace, and remove
# are defined earlier

jumpTable = {}
jumpTable['1'] = insert
jumpTable['2'] = replace
jumpTable['3'] = remove
```

Maintenance of the command processor becomes a matter of data management, wherein we add or remove entries in the jump table and the menu.

6.6 Exercises

- 1 Write the code for a mapping that generates a list of the absolute values of the numbers in a list named **numbers**.
- 2 Write the code for a filtering that generates a list of the positive numbers in a list named **numbers**. You should use a **lambda** to create the auxiliary function.
- 3 Write the code for a reducing that creates a single string from a list of strings named **words**.

- 4 Modify the **sum** function presented in Section 6.1 so that it includes default arguments for a step value and a function. The step value is used to move to the next value in the range. The function is applied to each number visited, and the function's returned value is added to the running total. The default step value is 1, and the default function is a **lambda** that returns its argument (essentially an identity function). An example call of this function is **sum(1, 100, 2, math.sqrt)**, which returns the sum of the square roots of every other number between 1 and 100. The function can also be called as usual, with just the bounds of the range.
- 5 Three versions of the summation function have been presented in this chapter. One uses a loop, one uses recursion, and one uses the **reduce** function. Discuss the costs and benefits of each version, in terms of programmer time and computational resources required.

Summary

- A function serves as an abstraction mechanism by allowing us to view many things as one thing.
- A function eliminates redundant patterns of code by specifying a single place where the pattern is defined.
- A function hides a complex chunk of code in a single named entity.
- A function allows a general method to be applied in varying situations. The variations are specified by the function's arguments.
- Functions support the division of labor when a complex task is factored into simpler subtasks.
- Top-down design is a strategy that decomposes a complex problem into simpler subproblems and assigns their solutions to functions. In top-down design, we begin with a top-level **main** function and gradually fill in the details of lower-level functions in a process of stepwise refinement.
- Cooperating functions communicate information by passing arguments and receiving return values. They also can receive information directly from common pools of data.
- A structure chart is a diagram of the relationships among cooperating functions. The chart shows the dependency relationships in a top-down design, as well as data flows among the functions and common pools of data.

- Recursive design is a special case of top-down design, in which a complex problem is decomposed into smaller problems of the same form. Thus, the original problem is solved by a single recursive function.
- A recursive function is a function that calls itself. A recursive function consists of at least two parts: a base case that ends the recursive process and a recursive step that continues it. These two parts are structured as alternative cases in a selection statement.
- The design of recursive algorithms and functions often follows the recursive character of a problem or a data structure.
- Although it is a natural and elegant problem-solving strategy, recursion can be computationally expensive. Recursive functions can require extra overhead in memory and processing time to manage the information used in recursive calls.
- An infinite recursion arises as the result of a design error. The programmer has not specified the base case or reduced the size of the problem in such a way that the termination of the process is reached.
- The namespace of a program is structured in terms of module variables, parameters, and temporary variables. A module variable, whether it names a function or a datum, is introduced and receives its initial value at the top level of the module. A parameter is introduced in a function header and receives its initial value when the function is called. A temporary variable is introduced in an assignment statement within the body of a function definition.
- The scope of a variable is the area of program text within which it has a given value. The scope of a module variable is the text of the module below the variable's introduction and the bodies of any function definitions. The scope of a parameter is the body of its function definition. The scope of a temporary variable is the text of the function body below its introduction.
- Scope can be used to control the visibility of names in a namespace. When two variables with different scopes have the same name, a variable's value is found by looking outward from the innermost enclosing scope. In other words, a temporary variable's value takes precedence over a parameter's value and a module variable's value when all three have the same name.
- The lifetime of a variable is the duration of program execution during which it uses memory storage. Module variables exist for the lifetime of the program that uses them. Parameters and temporary variables exist for the lifetime of a particular function call.

- Functions are first-class data objects. They can be assigned to variables, stored in data structures, passed as arguments to other functions, and returned as the values of other functions.
- Higher-order functions can expect other functions as arguments and/or return functions as values.
- A mapping function expects a function and a list of values as arguments. The function argument is applied to each value in the list and a map object containing the results is returned.
- A predicate is a Boolean function.
- A filtering function expects a predicate and a list of values as arguments. The values for which the predicate returns **True** are placed in a filter object and returned.
- A reducing function expects a function and a list of values as arguments. The function is applied to the values, and a single result is accumulated and returned.
- A jump table is a simple way to design a command processor. The table is a dictionary whose keys are command names and whose values are the associated functions. A function for a given command name is simply looked up in the table and called.

REVIEW QUESTIONS

- 1 Top-down design is a strategy that
 - a develops lower-level functions before the functions that depend on those lower-level functions
 - b starts with the **main** function and develops the functions on each successive level beneath the **main** function
- 2 The relationships among functions in a top-down design are shown in a
 - a syntax diagram
 - b flow diagram
 - c structure chart

- 3 A recursive function
- a usually runs faster than the equivalent loop
 - b usually runs more slowly than the equivalent loop
- 4 When a recursive function is called, the values of its arguments and its return address are placed in a
- a list
 - b dictionary
 - c set
 - d stack frame
- 5 The scope of a temporary variable is
- a the statements in the body of the function where the variable is introduced
 - b the entire module in which the variable is introduced
 - c the statements in the body of the function after the statement where the variable is introduced
- 6 The lifetime of a parameter is
- a the duration of program execution
 - b the duration of its function's execution
- 7 The expression `list(map(math.sqrt, [9, 25, 36]))` evaluates to
- a 70
 - b [81, 625, 1296]
 - c [3.0, 5.0, 6.0]
- 8 The expression `list(filter(lambda x: x > 50, [34, 65, 10, 100]))` evaluates to
- a []
 - b [65, 100]
- 9 The expression `reduce(max, [34, 21, 99, 67, 10])` evaluates to
- a 231
 - b 0
 - c 99

- 10 A data structure used to implement a jump table is a
- a list
 - b tuple
 - c dictionary

PROJECTS

- 1 Package Newton's method for approximating square roots (Case Study 3.6) in a function named **newton**. This function expects the input number as an argument and returns the estimate of its square root. The script should also include a **main** function that allows the user to compute square roots of inputs until she presses the enter/return key.
- 2 Convert Newton's method for approximating square roots in Project 1 to a recursive function named **newton**. (*Hint*: The estimate of the square root should be passed as a second argument to the function.)
- 3 Elena complains that the recursive **newton** function in Project 2 includes an extra argument for the estimate. The function's users should not have to provide this value, which is always the same, when they call this function. Modify the definition of the function so that it uses a keyword parameter with the appropriate default value for this argument, and call the function without a second argument to demonstrate that it solves this problem.
- 4 Restructure Newton's method (Case Study 3.6) by decomposing it into three cooperating functions. The **newton** function can use either the recursive strategy of Project 1 or the iterative strategy of Case Study 3.6. The task of testing for the limit is assigned to a function named **limitReached**, whereas the task of computing a new approximation is assigned to a function named **improveEstimate**. Each function expects the relevant arguments and returns an appropriate value.
- 5 A list is sorted in ascending order if it is empty or each item except the last one is less than or equal to its successor. Define a predicate **isSorted** that expects a list as an argument and returns **True** if the list is sorted, or returns **False** otherwise. (*Hint*: For a list of length 2 or greater, loop through the list and compare pairs of items, from left to right, and return **False** if the first item in a pair is greater.)

- 6 Add a command to this chapter's case study program that allows the user to view the contents of a file in the current working directory. When the command is selected, the program should display a list of filenames and a prompt for the name of the file to be viewed. Be sure to include error recovery.
- 7 Write a recursive function that expects a pathname as an argument. The pathname can be either the name of a file or the name of a directory. If the pathname refers to a file, its name is displayed, followed by its contents. Otherwise, if the pathname refers to a directory, the function is applied to each name in the directory. Test this function in a new program.
- 8 Lee has discovered what he thinks is a clever recursive strategy for printing the elements in a sequence (string, tuple, or list). He reasons that he can get at the first element in a sequence using the 0 index, and he can obtain a sequence of the rest of the elements by slicing from index 1. This strategy is realized in a function that expects just the sequence as an argument. If the sequence is not empty, the first element in the sequence is printed and then a recursive call is executed. On each recursive call, the sequence argument is sliced using the range **1:**. Here is Lee's function definition:

```
def printAll(seq):  
    if seq:  
        print(seq[0])  
        printAll(seq[1:])
```

Write a script that tests this function and add code to trace the argument on each call. Does this function work as expected? If so, explain how it actually works, and describe any hidden costs in running it.

- 9 Write a program that computes and prints the average of the numbers in a text file. You should make use of two higher-order functions to simplify the design.
- 10 Define and test a function **myRange**. This function should behave like Python's standard **range** function, with the required and optional arguments, but should return a list. Do not use the **range** function in your implementation! (*Hints:* Study Python's help on **range** to determine the names, positions, and what to do with your function's parameters. Use a default value of **None** for the two optional parameters. If these parameters both equal **None**, then the function has been called with just the stop value. If just the third parameter equals **None**, then the function has been called with a start value as well. Thus, the first part of the function's code establishes what the values of the parameters are or should be. The rest of the code uses those values to build a list by counting up or down.)

[CHAPTER] 7 SIMPLE GRAPHICS AND Image Processing

After completing this chapter, you will be able to:

- Use the concepts of object-based programming—classes, objects, and methods—to solve a problem
- Develop algorithms that use simple graphics operations to draw two-dimensional shapes
- Use the RGB system to create colors in graphics applications and modify pixels in images
- Develop recursive algorithms to draw recursive shapes
- Write a nested loop to process a two-dimensional grid
- Develop algorithms to perform simple transformations of images, such as conversion of color to grayscale

Until about 20 years ago, computers processed numbers and text almost exclusively. At the present time, the computational processing of images, video, and sound is becoming increasingly important. Computers have evolved from mere number crunchers and data processors to multimedia platforms utilizing a wide array of applications and devices, such as digital music players and digital cameras.

Ironically, all of these exciting tools and applications still rely on number crunching and data processing. However, because the supporting algorithms and data structures can be quite complex, they are often hidden from the average user. In this chapter, we explore

some basic concepts related to two important areas of media computing—graphics and image processing. We also examine **object-based programming**, a type of programming that relies on objects and methods to control complexity and solve problems in these areas.

7.1

Simple Graphics

Graphics is the discipline that underlies the representation and display of geometric shapes in two- and three-dimensional space. Python comes with a large array of resources that support graphics operations. However, these operations are complex and not for the faint of heart. To help you ease into the world of graphics, this section provides an introduction to a gentler set of graphics operations known as **Turtle graphics**. A Turtle graphics toolkit provides a simple and enjoyable way to draw pictures in a window and gives you an opportunity to run several methods with an object. In the next few sections, we use Python's **turtle** module to illustrate various features of object-based programming.

7.1.1

Overview of Turtle Graphics

Turtle graphics were originally developed as part of the children's programming language Logo, created by Seymour Papert and his colleagues at MIT in the late 1960s. The name is intended to suggest a way to think about the drawing process. Imagine a turtle crawling on a piece of paper with a pen tied to its tail. Commands direct the turtle as it moves across the paper and tell it to lift or lower its tail, turn some number of degrees left or right, and move a specified distance. Whenever the tail is down, the pen drags along the paper, leaving a trail. In this manner, it is possible to program the turtle to draw pictures ranging from the simple to the complex.

In the context of a computer, of course, the sheet of paper is a window on a display screen, and the turtle is an icon, such as an arrowhead. At any given moment in time, the turtle is located at a specific position in the window. This position is specified with (x, y) coordinates. The **coordinate system** for Turtle graphics is the standard Cartesian system, with the origin $(0, 0)$ at the center of a window. The turtle's initial position is the origin, which is also called the home.

In addition to its position, a turtle also has several other attributes, as described in Table 7.1.

Heading	Specified in degrees, the heading or direction increases in value as the turtle turns to the left, or counterclockwise. Conversely, a negative quantity of degrees indicates a right, or clockwise, turn. The turtle is initially facing east, or 0 degrees. North is 90 degrees.
Color	Initially black, the color can be changed to any of more than 16 million other colors.
Width	This is the width of the line drawn when the turtle moves. The initial width is 1 pixel. (You'll learn more about pixels shortly.)
Down	This attribute, which can be either true or false, controls whether the turtle's pen is up or down. When true (that is, when the pen is down), the turtle draws a line when it moves. When false (that is, when the pen is up), the turtle can move without drawing a line.

[TABLE 7.1] Some attributes of a turtle

Together, these attributes make up a turtle's state. The concept of state is a very important one in object-based programming. Generally, an object's state is the set of values of its attributes at any given point in time.

The turtle's state determines how the turtle will behave when any operations are applied to it. For example, a turtle will draw when it is moved if its pen is currently down, but it will simply move without drawing when its pen is currently up. Operations also change a turtle's state. For instance, moving a turtle changes its position, but not its direction, width, or color.

7.1.2 Turtle Operations

In Chapter 5, you learned that every data value in Python is actually an object. The types of objects are called classes. Included in a class are all of the methods (or operations) that apply to objects of that class. Because a turtle is an object, its operations are also defined as methods. Table 7.2 lists the methods of the **Turtle** class. In this table, the variable **t** refers to any particular **Turtle** object. Don't be concerned if you don't understand all the terms used in the table. You'll learn more about these graphics concepts throughout this chapter.

Turtle METHOD	WHAT IT DOES
<code>t = Turtle()</code>	Creates a new Turtle object and opens its window.
<code>t.home()</code>	Moves t to the center of the window and then points t east.
<code>t.up()</code>	Raises t 's pen from the drawing surface.
<code>t.down()</code>	Lowest t 's pen to the drawing surface.
<code>t.setheading(degrees)</code>	Points t in the indicated direction, which is specified in degrees. East is 0 degrees, north is 90 degrees, west is 180 degrees, and south is 270 degrees.
<code>t.left(degrees)</code> <code>t.right(degrees)</code>	Rotates t to the left or the right by the specified degrees.
<code>t.goto(x, y)</code>	Moves t to the specified position.
<code>t.forward(distance)</code>	Moves t the specified distance in the current direction.
<code>t.pencolor(r, g, b)</code> <code>t.pencolor(string)</code>	Changes the pen color of t to the specified RGB value or to the specified string, such as 'red' . Returns the current color of t when the arguments are omitted.
<code>t.fillcolor(r, g, b)</code> <code>t.fillcolor(string)</code>	Changes the fill color of t to the specified RGB value or to the specified string, such as 'red' . Returns the current fill color of t when the arguments are omitted.
<code>t.begin_fill()</code> <code>t.end_fill()</code>	Enclose a set of turtle commands that will draw a filled shape using the current fill color.
<code>t.width(pixels)</code>	Changes the width of t to the specified number of pixels. Returns t 's current width when the argument is omitted.
<code>t.hideturtle()</code> <code>t.showturtle()</code>	Makes the turtle invisible or visible.
<code>t.position()</code>	Returns the current position (x, y) of t .
<code>t.heading()</code>	Returns the current direction of t .
<code>t.isdown()</code>	Returns True if t 's pen is down or False otherwise.

[TABLE 7.2] The **Turtle** methods

The set of methods of a given class of objects make up its **interface**. This is another important idea in object-based programming. Programmers who use objects interact with them through their interfaces. Thus, an interface should contain all of the information necessary to use an object of a given class. This information includes method headers and documentation about the method's arguments, values returned, and changes to the state of the associated objects. As you have seen in previous chapters, Python's docstring mechanism allows the programmer to view an interface for an entire class or an individual method by entering expressions of the form `help(<class name>)` or `help(<class name>.<method name>)` at a shell prompt.

Now that you have the information necessary to use a **Turtle** object, let's define a function named **drawSquare**. This function expects a **Turtle** object, a pair of integers that indicate the coordinates of the square's upper-left corner, and an integer that designates the length of a side. The function begins by lifting the turtle up and moving it to the square's corner point. It then points the turtle due south—270 degrees—and places the turtle's pen down on the drawing surface. Finally, it moves the turtle the given length and turns it left by 90 degrees, four times. Here is the code for the **drawSquare** function:

```
def drawSquare(t, x, y, length):
    """Draws a square with the given turtle, an
    upper-left corner point (x, y), and a side's length."""
    t.up()
    t.goto(x, y)
    t.setheading(270)
    t.down()
    for count in range(4):
        t.forward(length)
        t.left(90)
```

As you can see, this function exercises half a dozen methods in the turtle's interface. Almost all you need to know in many graphics applications are the interfaces of the appropriate objects and the geometry of the desired shapes. Two other important classes used in Python's Turtle graphics system are **Screen**, which represents a turtle's associated window, and **Canvas**, which represents the area in which a turtle can move and draw lines. A canvas can be larger than its window, which displays just the area of the canvas visible to the human user. We will have more to say about these two objects later, but first let's examine how to create and manipulate a turtle.

7.1.3 Object Instantiation and the `turtle` Module

Before you apply any methods to an object, you must create the object. To be precise, you must create an **instance** of the object's class. The process of creating an object is called **instantiation**. In the programs you have seen so far in this book, Python automatically created objects such as numbers, strings, and lists when it encountered them as literals. The programmer must explicitly instantiate other classes of objects, including those that have no literals. The syntax for instantiating a class and assigning the resulting object to a variable is the following:

```
<variable name> = <class name>(<any arguments>)
```

The expression on the right side of the assignment, also called a **constructor**, resembles a function call. The constructor can receive as arguments any initial values for the new object's attributes, or other information needed to create the object. As you might expect, if the arguments are optional, reasonable defaults are provided automatically. The constructor then manufactures and returns a new instance of the class.

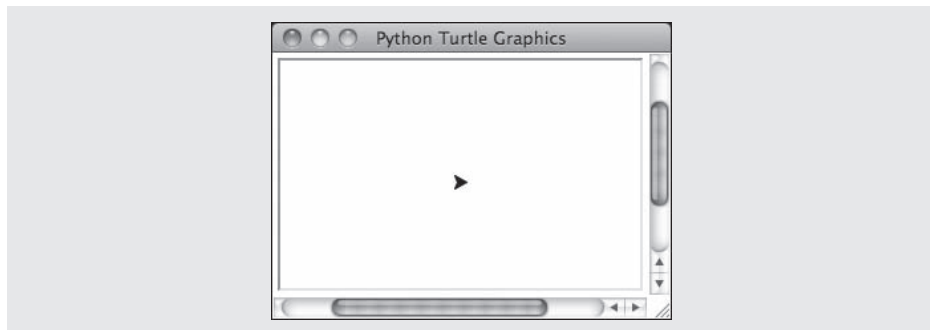
The **Turtle** class is defined in the **turtle** module (note carefully the spelling of both names). The following code then imports the **Turtle** class for use in a session (note that if you want to try this out yourself, you should skip to Section 7.1.10 and then return here):

```
>>> from turtle import Turtle
>>>
```

The next code segment creates and returns a **Turtle** object and opens a drawing window. The window is shown in Figure 7.1.

```
>>> t = Turtle()
```

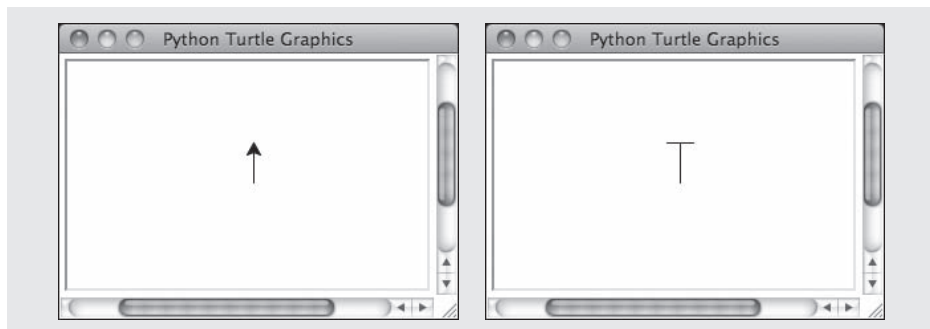
As you can see, the turtle's icon is located at the home position in the center of the window, facing east and ready to draw. The window is resizable; we show how to set the window's initial dimensions shortly.



[FIGURE 7.1] Drawing window for a turtle

Let's continue with the first turtle named `t`, and tell it to draw the letter T. It begins at the home position, turns 90 degrees left, and moves north 30 pixels to draw a vertical line. Then it turns 90 degrees left again to face west, picks its pen up, and moves 10 pixels. The turtle next turns to face due east, puts its pen down, and moves 20 pixels to draw a horizontal line. Finally, we hide the turtle. The session with the code follows. Figure 7.2 shows screenshots of the window after each line segment is drawn.

```
>>> t.left(90)           # Turn to face north
>>> t.forward(30)        # Draw vertical line
>>> t.left(90)           # Turn to face west
>>> t.up()               # Prepare to move without drawing
>>> t.forward(10)        # Move to beginning of horizontal line
>>> t.setheading(0)      # Turn to face east
>>> t.down()             # Prepare to draw
>>> t.forward(20)        # Draw horizontal line
>>> t.hideturtle()       # Make the turtle invisible
```



[FIGURE 7.2] Drawing vertical and horizontal lines for the letter T

To close a turtle's window, you click its close box. An attempt to manipulate a turtle whose window has been closed raises an error.

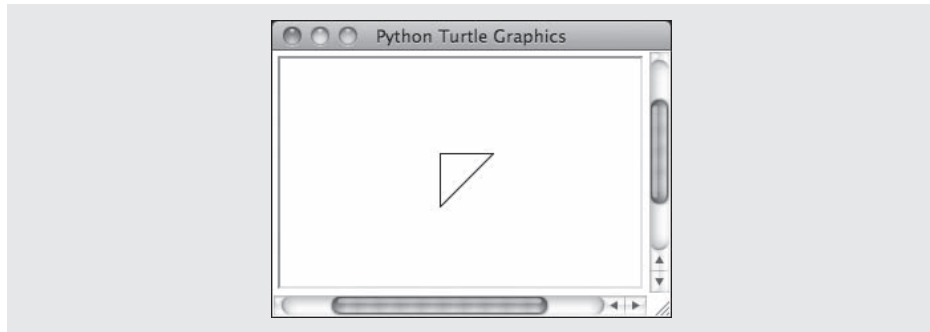
7.1.4 Drawing Two-Dimensional Shapes

Many graphics applications use **vector graphics**, or the drawing of simple two-dimensional shapes, such as rectangles, triangles, and circles. Most of these shapes can be represented as sets of vertices connected by line segments. For example, a triangle has three vertices and a pentagon has five vertices. Each vertex is a tuple of coordinates, and the set of vertices can be contained in a list. Using this information, you can define a **drawPolygon** Python function to draw most two-dimensional shapes. This function expects a **Turtle** object and a list of at least three vertices as arguments. The function raises the turtle's pen and moves it to the last vertex. The function then lowers the pen and moves the turtle to each vertex in the list, starting with the first one. The code for this function, followed by a call to draw a polygon, follows. A screenshot of the result is shown in Figure 7.3.

```
def drawPolygon(t, vertices):
    """Draws a polygon from a list of vertices.
    The list has the form [(x1, y1), ..., (xn, yn)]."""
    t.up()
    (x, y) = vertices[-1]
    t.goto(x, y)
    t.down()
    for (x, y) in vertices:
        t.goto(x, y)

>>> from turtle import Turtle
>>> t = Turtle()
>>> t.hideturtle()
>>> drawPolygon(t, [(20, 20), (-20, 20), (-20, -20)])
```

Note that the **for** loop in the **drawPolygon** function includes the tuple **(x, y)** where you would normally expect a single loop variable. This loop traverses a list of tuples, so on each pass through the loop, the variables **x** and **y** in the tuple **(x, y)** are assigned the corresponding values within the current tuple in the list.



[FIGURE 7.3] Drawing a polygon

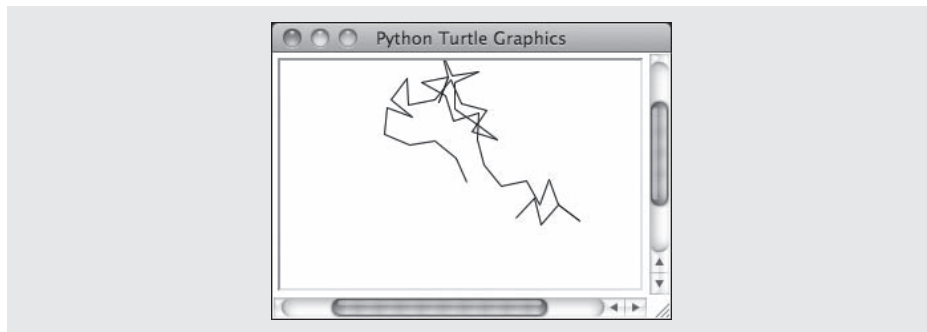
7.1.5 Taking a Random Walk

Animals often appear to wander about randomly, but they are often searching for food, shelter, a mate, and so forth. Or, they might be truly lost, disoriented, or just out for a stroll. Let's get a turtle to wander about randomly. A turtle engages in this harmless activity by repeatedly turning in a random direction and moving a given distance. The following script defines a function **randomWalk** that expects as arguments a **Turtle** object, the number of turns, and distance to move after each turn. The distance argument is optional and defaults to 20 pixels. When called in this script, the function performs 40 random turns with a distance of 40 pixels. Figure 7.4 shows one resulting output.

```
from turtle import Turtle
import random

def randomWalk(t, turns, distance = 20):
    """Turns a random number of degrees and moves
    a given distance for a fixed number of turns."""
    for x in range(turns):
        t.left(random.randint(0, 360))
        t.forward(distance)

randomWalk(Turtle(), 40)
```

[FIGURE 7.4] A random walk

7.1.6 Colors and the RGB System

The rectangular display area on a computer screen is made up of colored dots called picture elements or **pixels**. The smaller the pixel, the smoother the lines drawn with them will be. The size of a pixel is determined by the size and resolution of the display. For example, one common screen resolution is 1680 pixels by 1050 pixels, which, on a 20-inch monitor, produces a rectangular display area that is 17 inches by 10.5 inches. Setting the resolution to smaller values increases the size of the pixels, making the lines on the screen appear more ragged.

Each pixel represents a color. While the turtle's default color is black, you can easily change it to one of several other basic colors, such as red, yellow, or orange, by running the **pencolor** method with the corresponding string as an argument. For example, the following code segment changes the turtle's pen color (and the outline of the turtle's icon as well) to red:

```
>>> t.pencolor("red");
```

To provide the full range of several million colors available on today's computers, we need a more powerful representation scheme. Among the various schemes for representing colors, the **RGB system** is a fairly common one. The letters stand for the color components of red, green, and blue, to which the human retina is sensitive. These components are mixed together to form a unique color value. Naturally, the computer represents these values as integers, and the display hardware translates this information to the colors you see. Each color component can range from 0 through 255. The value 255 represents the maximum saturation of a given color component, whereas the value 0 represents the total absence of that component. Table 7.3 lists some example colors and their RGB values.

COLOR	RGB VALUE
Black	(0, 0, 0)
Red	(255, 0, 0)
Green	(0, 255, 0)
Blue	(0, 0, 255)
Yellow	(255, 255, 0)
Gray	(127, 127, 127)
White	(255, 255, 255)

[TABLE 7.3] Some example colors and their RGB values

You might be wondering how many total RGB color values are at your disposal. That number would be equal to all of the possible combinations of three values, each of which has 256 possible values, or $256 * 256 * 256$, or 16,777,216 distinct color values. Although the human eye cannot discriminate between adjacent color values in this set, the RGB system is called a **true color** system.

Another way to consider color is from the perspective of the computer memory required to represent a pixel's color. In general, N bits of memory can represent 2^N distinct data values. Conversely, N distinct data values require at least $\log_2 N$ bits of memory. In the old days, when memory was expensive and displays came in black and white, only a single bit of memory was required to represent the two color values. Thus, when displays capable of showing 8 shades of gray came along, 3 bits of memory were required to represent each color value. Early color monitors might have supported the display of 256 colors, so 8 bits were needed to represent each color value. Each color component of an RGB color requires 8 bits, so the total number of bits needed to represent a distinct color value is 24. The total number of RGB colors, 2^{24} , happens to be 16,777,216.

7.1.7 Example: Drawing with Random Colors

The **Turtle** class includes a **pencolor** method for changing the turtle's drawing color. This method expects integers for the three RGB components as arguments. The next script draws squares that are black, gray, and of two random colors at the corners of the turtle's window. The output is shown in Figure 7.5. (Note that the actual colors do not appear in this book.)

```

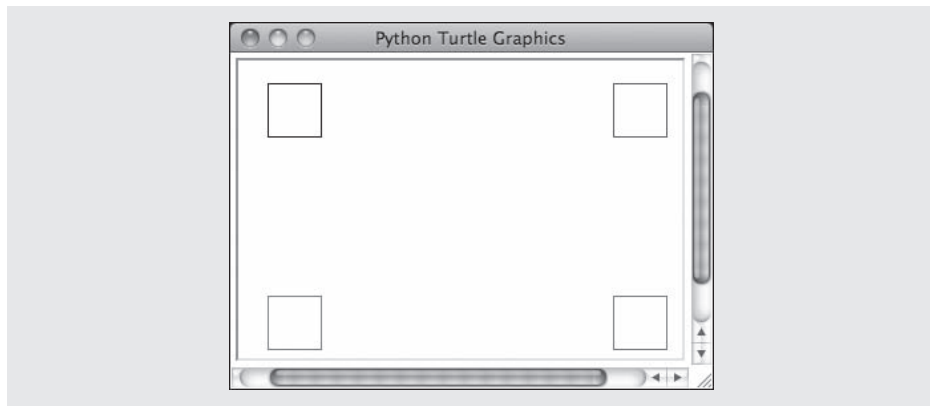
from turtle import Turtle
import random

def drawSquare(t, x, y, length):
    """ Draws a square with the upper-left corner (x, y)
    and the given length. """
    t.up()
    t.goto(x, y)
    t.setheading(270)
    t.down()
    for count in range(4):
        t.forward(length)
        t.left(90)

def main():
    t = Turtle()
    # Length of the square
    length = 40
    # Relative distances to corners of window from center
    width = t.screen.window_width() // 2
    height = t.screen.window_height() // 2
    # Draw in upper-left corner
    drawSquare(t, -width, height, length)
    # Gray
    t.pencolor(127, 127, 127)
    # Draw in lower-left corner
    drawSquare(t, -width, length - height, length)
    # First random color
    t.pencolor(random.randint(0, 255),
               random.randint(0, 255),
               random.randint(0, 255))
    # Draw in upper-right corner
    drawSquare(t, width - length, height, length)
    # Second random color
    t.pencolor(random.randint(0, 255),
               random.randint(0, 255),
               random.randint(0, 255))
    # Draw in lower-right corner
    drawSquare(t, width - length, length - height, length)

main()

```



[FIGURE 7.5] Four colored squares

7.1.8 Examining an Object's Attributes

The **Turtle** methods shown in the examples thus far modify a **Turtle** object's attributes, such as its position, heading, and color. These methods are called **mutator methods**, meaning that they change the internal state of a **Turtle** object. Other methods, such as **position()**, simply return the values of a **Turtle** object's attributes without altering its state. These methods are called **accessor methods**. The next code segment shows some accessor methods in action:

```
>>> from turtle import Turtle
>>> t = Turtle()
>>> t.position()
(0.0, 0.0)
>>> t.heading()
0.0
>>> t.isdown()
True
```

7.1.9 Manipulating a Turtle's Screen

As mentioned earlier, a **Turtle** object is associated with instances of the classes **Screen** and **Canvas**, which represent the turtle's window and the drawing area underneath it. The **Screen** object's attributes include its width and height in

pixels, and its background color, among other things. You access a turtle's **Screen** object using the notation **t.screen**, and then call a **Screen** method on this object. The methods **window_width()** and **window_height()** were used in an earlier example to locate the corners of a turtle's window. The following code resets the screen's background color, which is white by default, to orange:

```
>>> from turtle import Turtle
>>> t = Turtle()
>>> t.screen.bgcolor("orange")
```

7.1.10 Setting up a `cfg` File and Running IDLE

We have covered only a few of the commonly used methods to get you started with Turtle graphics programming. You can find a complete list with descriptions of their effects in the Python documentation. However, before you actually run a program with Python's **turtle** module, you need to set up a configuration file, and then launch IDLE in a slightly different manner than before.

A Turtle graphics configuration file, which has the file name **turtle.cfg**, is a text file that contains the initial settings of several attributes of **Turtle**, **Screen**, and **Canvas** objects. Python creates default settings for these attributes, which you can find in the Python documentation. For example, the default window size is $\frac{1}{2}$ of your computer monitor's width and $\frac{3}{4}$ of its height, and the window's title is "Python Turtle Graphics." If you want an initial window size of 300 by 200 pixels instead, you can override the default size by including the specific dimensions in a configuration file. The attributes in the file used for our examples are as follows:

```
width = 300
height = 200
using_IDLE = True
colormode = 255
```

To create a file with these settings, open a text editor, enter the settings as shown, and save the file as **turtle.cfg** in your current working directory (the one where you are saving your Python script files).

Now you must launch IDLE in a new way. Instead of double-clicking on its application icon from a window, you must open a terminal window (see Chapter 1) and navigate to your current working directory using the commands to change directories (either **cd pathname** or **cd ..**). From there, you run IDLE as a

command at the command prompt, with the **-n** option. For example, the following command launches IDLE for Python 3.1.2 in this manner:

```
> idle3.1 -n
```

At this point, you should see a shell window, as usual, and you should be able to run the Turtle graphics examples discussed in this section. Be sure, however, to open existing Python script files from this IDLE window's File menu, rather than launching them from your file system window.

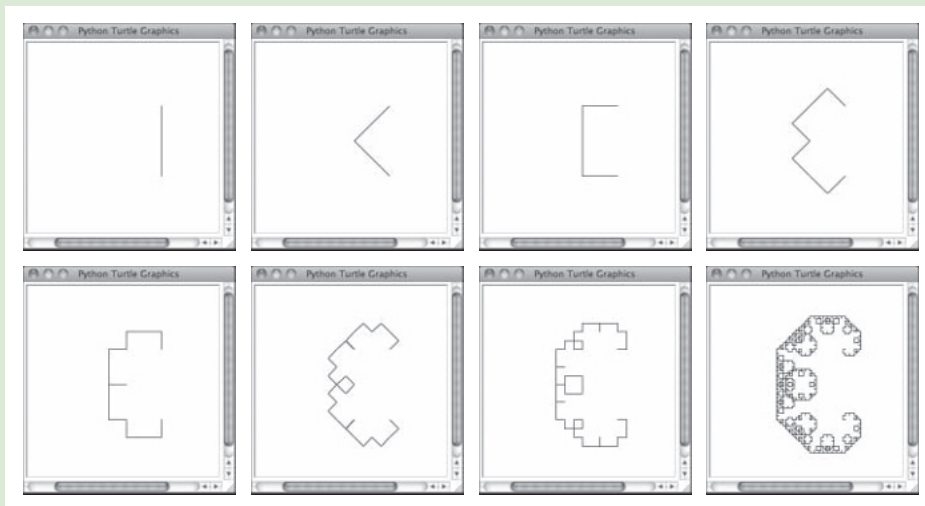
7.1 Exercises

- 1 Explain the importance of the interface of a class of objects.
- 2 What is object instantiation? What are the options at the programmer's disposal during this process?
- 3 Define a function named **drawLine**. This function expects a **Turtle** object and four integers as arguments. The integers represent the end-points of a line segment. The function should draw this line segment with the turtle and do no other drawing.
- 4 Describe what happens when you run the **pencolor()** method with a **Turtle** object.
- 5 Turtle graphics windows do not expand in size. What do you suppose happens when a **Turtle** object attempts to move beyond a window boundary?
- 6 Add arguments to the function **drawSquare** so that it uses these arguments to draw a square of a specified color.
- 7 The function **drawRectangle** expects a **Turtle** object and the coordinates of the upper-left and lower-right corners of a rectangle as arguments. Define this function, which draws the outline of the rectangle.
- 8 Define a **fillRectangle** function that takes the coordinates of a rectangle's upper-left and lower-right corner points and three integers representing an RGB value as arguments. The function should fill the rectangle in the given color. To fill a rectangle, you run the turtle's **pencolor** and **fillcolor** methods with the color, and place the code for drawing the rectangle between the method calls **t.begin_fill()** and **t.end_fill()**. You may call the **drawRectangle** function from Exercise 7 in your implementation to draw the rectangle.

7.2 Case Study: Recursive Patterns in Fractals

In this case study, we develop an algorithm that uses Turtle graphics to display a special kind of curve known as a **fractal object**. Fractals are highly repetitive or recursive patterns. A fractal object appears geometric, yet it cannot be described with ordinary Euclidean geometry. Strangely, a fractal curve is not one-dimensional, and a fractal surface is not two-dimensional. Instead, every fractal shape has its own fractal dimension. To understand what this means, let's start by considering the nature of an ordinary curve, which has a precise finite length between any two points. By contrast, a fractal curve has an indefinite length between any two points. The apparent length of a fractal curve depends on the level of detail in which it is viewed. As you zoom in on a segment of a fractal curve, you can see more and more details, and its length appears greater and greater. Consider a coastline, for example. Seen from a distance, it has many wiggles but a discernible length. Now put a piece of the coastline under magnification. It has many similar wiggles, and the discernible length increases. Self-similarity under magnification is the defining characteristic of fractals and is seen in the shapes of mountains, the branching patterns of tree limbs, and many other natural objects.

One example of a fractal curve is the **c-curve**. Figure 7.6 shows the first six levels of c-curves and a level-10 c-curve. The level-0 c-curve is a simple line segment. The level-1 c-curve replaces the level-0 c-curve with two smaller level-0 c-curves that meet at right angles. The level-2 c-curve does the same thing for each of the two line segments in the level-1 c-curve. This pattern of subdivision can continue indefinitely, producing quite intricate shapes. In the remainder of this case study, we develop an algorithm that uses Turtle graphics to display a c-curve.



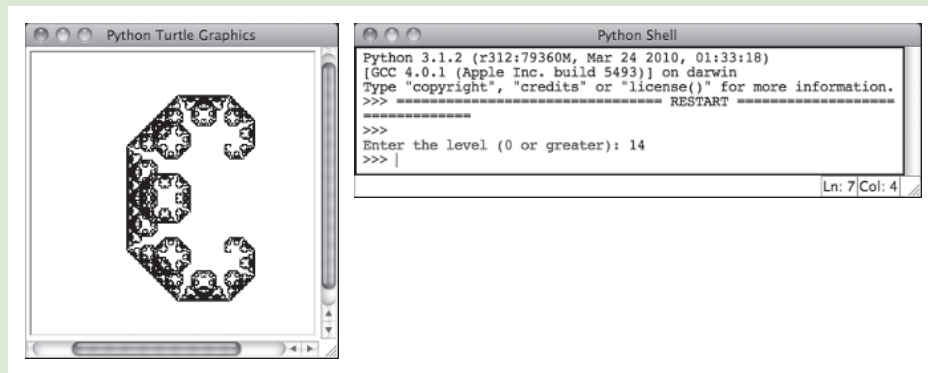
[FIGURE 7.6] C-curves of levels 0 through 6 and a c-curve of level 10

7.2.1 Request

Write a program that allows the user to draw a particular c-curve in varying degrees.

7.2.2 Analysis

The proposed interface is shown in Figure 7.7. The program should prompt the user for the level of the c-curve. After this integer is entered, the program should display a Turtle graphics window in which it draws the c-curve.



[FIGURE 7.7] The interface for the c-curve program

7.2.3 Design

An N -level c-curve can be drawn with a recursive function. The function receives a **Turtle** object, the end points of a line segment, and the current level as arguments. At level 0, the function draws a simple line segment. Otherwise, a level N c-curve consists of two level $N - 1$ c-curves, constructed as follows:

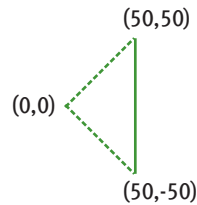
Let xm be $(x1 + x2 + y1 - y2) // 2$.

Let ym be $(x2 + y1 + y2 - x1) // 2$.

The first level $N - 1$ c-curve uses the line segment $(x1, y1)$, (xm, ym) , and level $N - 1$, so the function is called recursively with these arguments.

The second level $N - 1$ c-curve uses the line segment (xm, ym) , $(x2, y2)$, and level $N - 1$, so the function is called recursively with these arguments.

For example, in a level-0 c-curve, let (x_1, y_1) be $(50, -50)$ and (x_2, y_2) be $(50, 50)$. Then, to obtain a level-1 c-curve, use the formulas for computing x_m and y_m to obtain (x_m, y_m) , which is $(0, 0)$. Figure 7.8 shows a solid line segment for the level-0 c-curve and two dashed line segments for the level-1 c-curve that result from these operations. In effect, the operations produce two shorter line segments that meet at right angles.



[FIGURE 7.8] A level-0 c-curve (solid) and a level-1 c-curve (dashed)

Here is the pseudocode for the recursive algorithm:

```
function cCurve(t, x1, y1, x2, y2, level)
    if level == 0:
        drawLine(x1, y1, x2, y2)
    else
        xm = (x1 + x2 + y1 - y2) // 2
        ym = (x2 + y1 + y2 - x1) // 2
        cCurve(t, x1, y1, xm, ym, level - 1)
        cCurve(t, xm, ym, x2, y2, level - 1)
```

The function **drawLine** uses the turtle to draw a line between two given endpoints.

7.2.4 Implementation (Coding)

The program includes the three function definitions of **cCurve**, **drawLine**, and **main**. Because **drawLine** is an auxiliary function, its definition is nested within the definition of **cCurve**.

```
"""
Program file: ccurve.py
Author: Ken

This program prompts the user for the level of
a c-curve and draws a c-curve of that level.
"""

from turtle import Turtle

def cCurve(t, x1, y1, x2, y2, level):
    """Draws a c-curve of the given level."""

    def drawLine(x1, y1, x2, y2):
        """Draws a line segment between the endpoints."""
        t.up()
        t.goto(x1, y1)
        t.down()
        t.goto(x2, y2)

    if level == 0:
        drawLine(x1, y1, x2, y2)
    else:
        xm = (x1 + x2 + y1 - y2) // 2
        ym = (x2 + y1 + y2 - x1) // 2
        cCurve(t, x1, y1, xm, ym, level - 1)
        cCurve(t, xm, ym, x2, y2, level - 1)

def main():
    level = int(input("Enter the level (0 or greater): "))
    t = Turtle()
    t.hideturtle()
    cCurve(t, 50, -50, 50, 50, level)

main()
```

7.3 Image Processing

Over the centuries, human beings have developed numerous technologies for representing the visual world, the most prominent being sculpture, painting, photography, and motion pictures. The most recent form of this type of technology is digital image processing. This enormous field includes the principles and techniques for the following:

- The capture of images with devices such as flatbed scanners and digital cameras
- The representation and storage of images in efficient file formats
- Constructing the algorithms in image-manipulation programs such as Adobe Photoshop

In this section, we focus on some of the basic concepts and principles used to solve problems in image processing.

7.3.1 Analog and Digital Information

Representing photographic images in a computer poses an interesting problem. As you have seen, computers must use digital information which consists of **discrete values**, such as individual integers, characters of text, or bits in a bit string. However, the information contained in images, sound, and much of the rest of the physical world is analog. **Analog information** contains a **continuous range** of values. You can get an intuitive sense of what this means by contrasting the behaviors of a digital clock and a traditional analog clock. A digital clock shows each second as a discrete number on the display. An analog clock displays the seconds as tick marks on a circle. The clock's second hand passes by these marks as it sweeps around the clock's face. This sweep reveals the analog nature of time: between any two tick marks on the analog clock, there is a continuous range of positions or moments of time through which the second hand passes. You can represent these moments as fractions of a second, but between any two such moments are others that are more precise (recall the concept of precision used with real numbers). The ticks representing seconds on the analog clock's face thus represent an attempt to **sample** moments of time as discrete values, whereas time itself is continuous, or analog.

Early recording and playback devices for images and sound were all analog devices. If you examine the surface of a vinyl record under a magnifying glass, you will notice grooves with regular wave patterns. These patterns directly reflect, or analogize, the continuous wave forms of the recorded sounds.

Likewise, the chemical media on photographic film directly reflect the continuous color and intensity values of light reflected from the subjects of photographs.

Somehow, the continuous analog information in a real visual scene must be mapped into a set of discrete values. This conversion process also involves sampling, a technology we consider next.

7.3.2 Sampling and Digitizing Images

A visual scene projects an infinite set of color and intensity values onto a two-dimensional sensing medium, such as a human being's retina or a scanner's surface. If you sample enough of these values, the digital information can represent an image that is more or less indistinguishable to the human eye from the original scene.

Sampling devices measure discrete color values at distinct points on a two-dimensional grid. These values are pixels, which were introduced earlier in this chapter. In theory, the more pixels that are sampled, the more continuous and realistic the resulting image will appear. In practice, however, the human eye cannot discern objects that are closer together than 0.1 mm, so a sampling of 10 pixels per linear millimeter (250 pixels per inch and 62,500 pixels per square inch) would be plenty accurate. Thus, a 3-inch by 5-inch image would need

$$3 * 5 * 62,500 \text{ pixels/inch}^2 = 937,500 \text{ pixels}$$

which is approximately one megapixel. For most purposes, however, you can settle for a much lower sampling size and, thus, fewer pixels per square inch.

7.3.3 Image File Formats

Once an image has been sampled, it can be stored in one of many file formats. A **raw image file** saves all of the sampled information. This has a cost and a benefit: the benefit is that the display of a raw image will be the most true to life, but the cost is that the file size of the image can be quite large. Back in the days when disk storage was still expensive, computer scientists developed several schemes to compress the data of an image to minimize its file size. Although storage is now cheap, these formats are still quite economical for sending images across networks. Two of the most popular image file formats are JPEG (Joint Photographic Experts Group) and GIF (Graphic Interchange Format).

Various data-compression schemes are used to reduce the file size of a JPEG image. One scheme examines the colors of each pixel's neighbors in the grid. If any color values are the same, their positions rather than their values are stored,

thus potentially saving many bits of storage. Before the image is displayed, the original color values are restored during the process of decompression. This scheme is called **lossless compression**, meaning that no information is lost. To save even more bits, another scheme analyzes larger regions of pixels and saves a color value that the pixels' colors approximate. This is called a **lossy scheme**, meaning that some of the original color information is lost. However, when the image is decompressed and displayed, the human eye usually is not able to detect the difference between the new colors and the original ones.

A GIF image relies on an entirely different compression scheme. The compression algorithm consists of two phases. In the first phase, the algorithm analyzes the color samples to build a table, or **color palette**, of up to 256 of the most prevalent colors. The algorithm then visits each sample in the grid and replaces it with the *key* of the closest color in the color palette. The resulting image file thus consists of at most 256 color values and the integer keys of the image's colors in the palette. This strategy can potentially save a huge number of bits of storage. The decompression algorithm uses the keys and the color palette to restore the grid of pixels for display. Although GIF uses a lossy compression scheme, it works very well for images with broad, flat areas of the same color, such as cartoons, backgrounds, and banners.

7.3.4 Image-Manipulation Operations

Image-manipulation programs either transform the information in the pixels or alter the arrangement of the pixels in the image. These programs also provide fairly low-level operations for transferring images to and from file storage. Among other things, these programs can do the following:

- Rotate an image
- Convert an image from color to grayscale
- Apply color filtering to an image
- Highlight a particular area in an image
- Blur all or part of an image
- Sharpen all or part of an image
- Control the brightness of an image
- Perform edge detection on an image
- Enlarge or reduce an image's size
- Apply color inversion to an image
- Morph an image into another image

You'll learn how to write Python code that can perform some of these manipulation tasks later in this chapter, and you will have a chance to practice others in the programming projects.

7.3.5 The Properties of Images

When an image is loaded into a program such as a Web browser, the software maps the bits from the image file into a rectangular area of colored pixels for display. The coordinates of the pixels in this two-dimensional grid range from (0, 0) at the upper-left corner of an image to (*width* - 1, *height* - 1) at the lower-right corner, where *width* and *height* are the image's dimensions in pixels. Thus, the **screen coordinate system** for the display of an image is somewhat different from the standard Cartesian coordinate system that we used with Turtle graphics, where the origin (0,0) is at the center of the rectangular grid. The RGB color system introduced earlier in this chapter is a common way of representing the colors in images. For our purposes, then, an image consists of a width, a height, and a set of color values accessible by means of (x, y) coordinates. A color value consists of the tuple (*r*, *g*, *b*), where the variables refer to the integer values of its red, green, and blue components, respectively.

7.3.6 The `images` Module

To facilitate our discussion of image-processing algorithms, we now present a small module of high-level Python resources for image processing. This package of resources, which is named **`images`**, allows the programmer to load an image from a file, view the image in a window, examine and manipulate an image's RGB values, and save the image to a file. The **`images`** module is a non-standard, open-source Python tool. You can find installation instructions in Appendix B, but placing the file **`images.py`** and some sample image files in your current working directory will get you started.

The **`images`** module includes a class named **`Image`**. The **`Image`** class represents an image as a two-dimensional grid of RGB values. The methods for the **`Image`** class are listed in Table 7.4. In this table, the variable **`i`** refers to an instance of the **`Image`** class.

Image METHOD	WHAT IT DOES
<code>i = Image(filename)</code>	Loads and returns an image from a file with the given filename. Raises an error if the filename is not found or the file is not a GIF file.
<code>i = Image(width, height)</code>	Creates and returns a blank image with the given dimensions. The color of each pixel is white, and the filename is the empty string.
<code>i.getWidth()</code>	Returns the width of <code>i</code> in pixels.
<code>i.getHeight()</code>	Returns the height of <code>i</code> in pixels.
<code>i.getPixel(x, y)</code>	Returns a tuple of integers representing the RGB values of the pixel at position (x, y).
<code>i.setPixel(x, y, (r, g, b))</code>	Replaces the RGB value at the position (x, y) with the RGB value given by the tuple (r, g, b).
<code>i.draw()</code>	Displays <code>i</code> in a window. The user must close the window to return control to the method's caller.
<code>i.clone()</code>	Returns a copy of <code>i</code> .
<code>i.save()</code>	Saves <code>i</code> under its current filename. If <code>i</code> does not yet have a filename, <code>save</code> does nothing.
<code>i.save(filename)</code>	Saves <code>i</code> under <code>filename</code> . Automatically adds a <code>.gif</code> extension if <code>filename</code> does not contain it.

[TABLE 7.4] The `Image` methods

Before we discuss some standard image-processing algorithms, let's try out the resources of the `images` module. This version of the `images` module accepts only image files in GIF format. For the purposes of this exercise, we also assume that a GIF image of my cat, Smokey, has been saved in a file named `smokey.gif` in the current working directory. The following session with the interpreter does three things:

- 1 Imports the `Image` class from the `images` module
- 2 Instantiates this class using the file named `smokey.gif`
- 3 Draws the image

The resulting image display window is shown in Figure 7.9. Although the actual image is in color, with green grass surrounding the cat, in this book the colors are not visible.


```
>>> from images import Image
>>> image = Image("smokey.gif")
>>> image.draw()
```



[FIGURE 7.9] An image display window

Python raises an error if it cannot locate the file in the current directory, or if the file is not a GIF file. Note also that the user must close the window to return control to the caller of the method **draw**. If you are working in the shell, the shell prompt will reappear when you do this. The image can then be redrawn, after other operations are performed, by calling **draw** again.

Once an image has been created, you can examine its width and height, as follows:

```
>>> image.getWidth()
198
>>> image.getHeight()
149
>>>
```

Alternatively, you can print the image's string representation:

```
>>> print(image)
Filename: smokey.gif
Width: 198
Height: 149
>>>
```

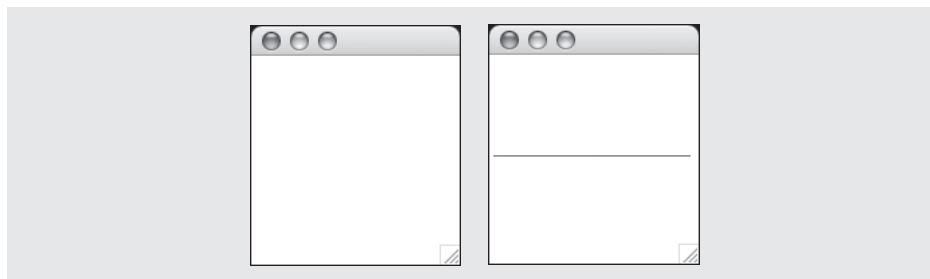
The method **getPixel** returns a tuple of the RGB values at the given coordinates. The following session shows the information for the pixel at position (0, 0), which is at the image's upper-left corner.

```
>>> image.getPixel(0, 0)
(194, 221, 114)
```

Instead of loading an existing image from a file, the programmer can create a new, blank image. The programmer specifies the image's width and height; the resulting image consists of all white pixels. Such images are useful for creating backgrounds or drawing simple shapes, or creating new images that receive information from existing images.

The programmer can use the method **setPixel** to replace an RGB value at a given position in an image. The next session creates a new 150 by 150 image. The pixels along a horizontal line at the middle of the image are then replaced with new blue pixels. The images before and after this transformation are shown in Figure 7.10. The loop visits every pixel along the row of pixels whose *y* coordinate is the image's height divided by 2.

```
>>> image = Image(150, 150)
>>> image.draw()
>>> blue = (0, 0, 255)
>>> y = image.getHeight() // 2
>>> for x in range(image.getWidth()):
>>>     image.setPixel(x, y, blue)
>>> image.draw()
```



[FIGURE 7.10] An image before and after replacing the pixels

Finally, you can save an image under its current filename or a different filename. Use the **save** operation to write an image back to an existing file using the current filename. The **save** operation can also receive a string parameter for a new filename. The image is written to a file with that name, which then becomes the current filename. The following code saves the new image using the filename **horizontal.gif**:

```
>>> image.save("horizontal.gif")
```

If you omit the **.gif** extension in the filename, the method adds it automatically.

7.3.7 A Loop Pattern for Traversing a Grid

Most of the loops we have used in this book have had a **linear loop structure**—that is, they visit each element in a sequence or they count through a sequence of numbers using a single loop control variable. By contrast, many image-processing algorithms use a **nested loop structure** to traverse a two-dimensional grid of pixels. Figure 7.11 shows such a grid. Its height is 3 rows, numbered 0 through 2. Its width is 5 columns, numbered 0 through 4. Each data value in the grid is accessed with a pair of coordinates using the form (**<column>**, **<row>**). Thus, the datum in the middle of the grid, which is shaded, is at position (2, 1). The datum in the upper-left corner is at the origin of the grid, (0, 0).

	0	1	2	3	4
0					
1					
2					

[FIGURE 7.11] A grid with 3 rows and 5 columns

A nested loop structure to traverse a grid consists of two loops, an outer one and an inner one. Each loop has a different loop control variable. The outer loop iterates over one coordinate, while the inner loop iterates over the other coordinate.

Here is a session that prints the pairs of coordinates visited when the outer loop traverses the *y* coordinates:

```
>>> width = 2
>>> height = 3
>>> for y in range(height):
    for x in range(width):
        print((x, y))
    print()

(0, 0) (1, 0)
(0, 1) (1, 1)
(0, 2) (1, 2)
>>>
```

As you can see, this loop marches across a row in an imaginary 2 by 3 grid, prints the coordinates at each column in that row, and then moves on to the next row. The following template captures this pattern, which is called a **row-major traversal**. We use this template to develop many of the algorithms that follow.

```
for y in range(height):
    for x in range(width):
        do something at position (x, y)
```

7.3.8 A Word on Tuples

Many of the algorithms obtain a pixel from the image, apply some function to the pixel's RGB values, and reset the pixel with the results. Because a pixel's RGB values are stored in a tuple, manipulating them is quite easy. Python allows the assignment of one tuple to another in such a manner that the elements of the source tuple can be bound to distinct variables in the destination tuple. For example, suppose you want to increase each of a pixel's RGB values by 10, thereby making the pixel brighter. You first call `getPixel` to retrieve a tuple and assign it to a tuple that contains three variables, as follows:

```
>>> (r, g, b) = image.getPixel(0, 0)
```

You can now see what the RGB values are by examining the following variables:

```
>>> r
194
>>> g
221
>>> b
114
```

The task is completed by building a new tuple with the results of the computations and resetting the pixel to that tuple:

```
>>> image.setPixel(0, 0, (r + 10, g + 10, b + 10))
```

The elements of a tuple cannot be bound to variables when that tuple is passed as an argument to a function. For example, the function **average** computes the average of the numbers in a 3-tuple as follows:

```
>>> def average(triple):
    (a, b, c) = triple
    return (a + b + c) // 3

>>> average((40, 50, 60))
50
>>>
```

Armed with these basic operations, we can now examine some simple image-processing algorithms. Some of the algorithms visit every pixel in an image and modify its color in some manner. Other algorithms use the information from an image's pixels to build a new image. For consistency and ease of use, we represent each algorithm as a Python function that expects an image as an argument. Some functions return a new image, whereas others simply modify the argument image.

7.3.9 Converting an Image to Black and White

Perhaps the easiest transformation is to convert a color image to black and white. For each pixel, the algorithm computes the average of the red, green, and blue values. The algorithm then resets the pixel's color values to 0 (black) if the average is closer to 0, or to 255 (white) if the average is closer to 255. The code for the function **blackAndWhite** follows. Figure 7.12 shows Smokey the cat before

and after the transformation. (Keep in mind that the original image is a color image; the colors are not visible in this book.)

```
def blackAndWhite(image):  
    """Converts the argument image to black and white."""  
    blackPixel = (0, 0, 0)  
    whitePixel = (255, 255, 255)  
    for y in range(image.getHeight()):  
        for x in range(image.getWidth()):  
            (r, g, b) = image.getPixel(x, y)  
            average = (r + g + b) // 3  
            if average < 128:  
                image.setPixel(x, y, blackPixel)  
            else:  
                image.setPixel(x, y, whitePixel)
```



[FIGURE 7.12] Converting a color image to black and white

Note that the second image appears rather stark, like a woodcut. The function can be tested in a short script, as follows:

```
from images import Image  
  
# Code for blackAndWhite's function definition goes here  
  
def main(filename = "smokey.gif"):  
    image = Image(filename)  
    print("Close the image window to continue. ")  
    image.draw()  
    blackAndWhite(image)  
    print("Close the image window to quit. ")  
    image.draw()  
  
main()
```

Note that the **main** function includes an optional argument for the image filename. Its default should be the name of an image in the current working directory.

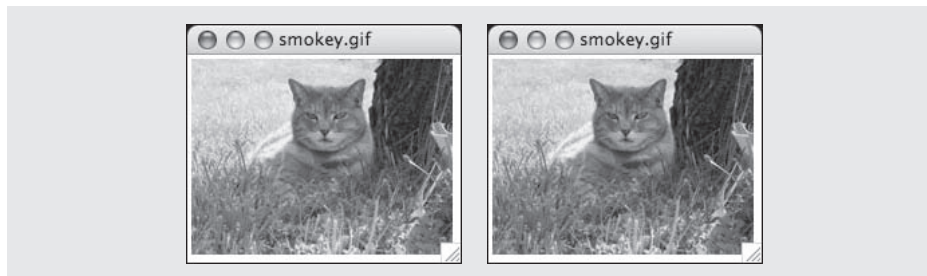
7.3.10 Converting an Image to Grayscale

Black-and-white photographs are not really just black and white, but also contain various shades of gray known as **grayscale**. (In fact, the original color images of Smokey the cat, which you saw earlier in this chapter, are reproduced in grayscale in this book.) Grayscale can be an economical color scheme, wherein the only color values might be 8, 16, or 256 shades of gray (including black and white at the extremes). Let's consider how to convert a color image to grayscale. As a first step, you might try replacing the color values of each pixel with their average, as follows:

```
average = (r + g + b) // 3
image.setPixel(x, y, (average, average, average))
```

Although this method is simple, it does not reflect the manner in which the different color components affect human perception. The human eye is actually more sensitive to green and red than it is to blue. As a result, the blue component appears darker than the other two components. A scheme that combines the three components needs to take these differences in **luminance** into account. A more accurate method would weight green more than red and red more than blue. Therefore, to obtain the new RGB values, instead of adding up the color values and dividing by 3, you should multiply each one by a weight factor and add the results. Psychologists have determined that the relative luminance proportions of green, red, and blue are .587, .299, and .114, respectively. Note that these values add up to 1. The next function, **grayscale**, uses this strategy, and Figure 7.13 shows the results.

```
def grayscale(image):
    """Converts the argument image to grayscale."""
    for y in range(image.getHeight()):
        for x in range(image.getWidth()):
            (r, g, b) = image.getPixel(x, y)
            r = int(r * 0.299)
            g = int(g * 0.587)
            b = int(b * 0.114)
            lum = r + g + b
            image.setPixel(x, y, (lum, lum, lum))
```



[FIGURE 7.13] Converting a color image to grayscale

A comparison of the results of this algorithm with those of the simpler one using the crude averages is left as an exercise for you.

7.3.11 Copying an Image

The next few algorithms do not modify an existing image, but instead use that image to generate a brand new image with the desired properties. One could create a new, blank image of the same height and width as the original, but it is often useful to start with an exact copy of the original image that retains the pixel information as well. The **Image** class includes a **clone** method for this purpose. The method **clone** builds and returns a new image with the same attributes as the original one, but with an empty string as the filename. The two images are thus structurally equivalent but not identical, as discussed in Chapter 5. This means that changes to the pixels in one image will have no impact on the pixels in the same positions in the other image. The following session demonstrates the use of the **clone** method:

```
>>> from images import Image
>>> image = Image("smokey.gif")
>>> image.draw()
>>> newImage = image.clone()      # Create a copy of image
>>> newImage.draw()
>>> grayscale(newImage)          # Change in second window only
>>> newImage.draw()
>>> image.draw()
```


7.3.12 Blurring an Image

Occasionally, an image appears to contain rough, jagged edges. This condition, known as **pixilation**, can be mitigated by blurring the image's problem areas. **Blurring** makes these areas appear softer, but at the cost of losing some definition. We now develop a simple algorithm to blur an entire image. This algorithm resets each pixel's color to the average of the colors of the four pixels that surround it. The function **blur** expects an image as an argument and returns a copy of that image with blurring. The function **blur** begins its traversal of the grid with position (1, 1) and ends with position (*width* - 2, *height* - 2). Although this means that the algorithm does not transform the pixels on the image's outer edges, you do not have to check for the grid's boundaries when you obtain information from a pixel's neighbors. Here is the code for **blur**, followed by an explanation:

```
def blur(image):
    """Builds and returns a new image which is a blurred
    copy of the argument image."""

    def tripleSum(triple1, triple2):          #1
        (r1, g1, b1) = triple1
        (r2, g2, b2) = triple2
        return (r1 + r2, g1 + g2, b1 + b2)

    new = image.clone()
    for y in range(1, image.getHeight() - 1):
        for x in range(1, image.getWidth() - 1):
            oldP = image.getPixel(x, y)
            left = image.getPixel(x - 1, y)   # To left
            right = image.getPixel(x + 1, y)  # To right
            top = image.getPixel(x, y - 1)    # Above
            bottom = image.getPixel(x, y + 1) # Below
            sums = reduce(tripleSum,          #2
                          [oldP, left, right, top, bottom])
            averages = tuple(map(lambda x: x / 5, sums)) #3
            new.setPixel(x, y, averages)
    return new
```

The code for **blur** includes some interesting design work. In the following explanation, the numbers noted appear to the right of the corresponding lines of code:

- At **#1**, the nested auxiliary function **tripleSum** is defined. This function expects two tuples of integers as arguments and returns a single tuple containing the sums of the values at each position.

- At #2, five tuples of RGB values are wrapped in a list and passed with the **tripleSum** function to the **reduce** function. This function repeatedly applies **tripleSum** to compute the sums of the tuples, until a single tuple containing the sums is returned.
- At #3, a **lambda** function is mapped onto the tuple of sums, and the resulting list is converted to a tuple. The **lambda** function divides each sum by 5. Thus, you are left with a tuple of the average RGB values.

Although this code is still rather complex, try writing it without **map** and **reduce**, and then compare the two versions.

7.3.13 Edge Detection

When artists paint pictures, they often sketch an outline of the subject in pencil or charcoal. They then fill in and color over the outline to complete the painting. **Edge detection** performs the inverse function on a color image: it removes the full colors to uncover the outlines of the objects represented in the image.

A simple edge-detection algorithm examines the neighbors below and to the left of each pixel in an image. If the luminance of the pixel differs from that of either of these two neighbors by a significant amount, you have detected an edge, and you set that pixel's color to black. Otherwise, you set the pixel's color to white.

The function **detectEdges** expects an image and an integer as parameters. The function returns a new black-and-white image that explicitly shows the edges in the original image. The integer parameter allows the user to experiment with various differences in luminance. Figure 7.14 shows the image of Smokey the cat before and after detecting edges with luminance thresholds of 10 and 20. Here is the code for function **detectEdges**:

```
def detectEdges(image, amount):
    """Builds and returns a new image in which the
    edges of the argument image are highlighted and
    the colors are reduced to black and white."""

    def average(triple):
        (r, g, b) = triple
        return (r + g + b) // 3

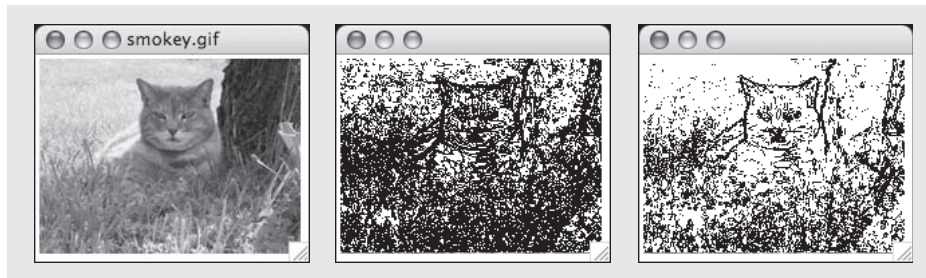
    blackPixel = (0, 0, 0)
    whitePixel = (255, 255, 255)
    new = image.clone()
```

continued

```

for y in range(image.getHeight() - 1):
    for x in range(1, image.getWidth()):
        oldPixel = image.getPixel(x, y)
        leftPixel = image.getPixel(x - 1, y)
        bottomPixel = image.getPixel(x, y + 1)
        oldLum = average(oldPixel)
        leftLum = average(leftPixel)
        bottomLum = average(bottomPixel)
        if abs(oldLum - leftLum) > amount or \
            abs(oldLum - bottomLum) > amount:
            new.setPixel(x, y, blackPixel)
        else:
            new.setPixel(x, y, whitePixel)
return new

```



[FIGURE 7.14] Edge detection: the original image, a luminance threshold of 10, and a luminance threshold of 20

7.3.14 Reducing the Image Size

The size and the quality of an image on a display medium, such as a computer monitor or a printed page, depend on two factors: the image's width and height in pixels and the display medium's **resolution**. Resolution is measured in pixels, or dots per inch (DPI). When the resolution of a monitor is increased, the images appear smaller, but their quality increases. Conversely, when the resolution is decreased, images become larger, but their quality degrades. Some devices, such as printers, provide good-quality image displays with small DPIs such as 72, whereas monitors tend to give better results with higher DPIs. You can set the resolution of an image itself before the image is captured. Scanners and digital cameras have controls that allow the user to specify the DPI values. A higher DPI causes the sampling device to take more samples (pixels) through the two-dimensional grid.

In this section, we ignore the issues raised by resolution and learn how to reduce the size of an image once it has been captured. (For the purposes of this

discussion, the size of an image is its width and height in pixels.) Reducing an image's size can dramatically improve its performance characteristics, such as load time in a Web page and space occupied on a storage medium. In general, if the height and width of an image are each reduced by a factor of N , the number of color values in the resulting image is reduced by a factor of N^2 .

A size reduction usually preserves an image's **aspect ratio** (that is, the ratio of its width to its height). A simple way to shrink an image is to create a new image whose width and height are a constant fraction of the original image's width and height. The algorithm then copies the color values of just some of the original image's pixels to the new image. For example, to reduce the size of an image by a factor of 2, you could copy the color values from every other row and every other column of the original image to the new image.

The Python function **shrink** exploits this strategy. The function expects the original image and a positive integer shrinkage factor as parameters. A shrinkage factor of 2 tells Python to shrink the image to $1/2$ of its original dimensions, a factor of 3 tells Python to shrink the image to $1/3$ of its original dimensions, and so forth. The algorithm uses the shrinkage factor to compute the size of the new image and then creates it. Because a one-to-one mapping of grid positions in the two images is not possible, separate variables are used to track the positions of the pixels in the original image and the new image. The loop traverses the larger image (the original) and skips positions by incrementing its coordinates by the shrinkage factor. The new image's coordinates are incremented by 1, as usual. The loop continuation conditions are also offset by the shrinkage factor to avoid range errors. Here is the code for the function **shrink**:

```
def shrink(image, factor):
    """Builds and returns a new image which is a smaller
    copy of the argument image, by the factor argument."""
    width = image.getWidth()
    height = image.getHeight()
    new = Image(width // factor, height // factor)
    oldY = 0
    newY = 0
    while oldY < height - factor:
        oldX = 0
        newX = 0
        while oldX < width - factor:
            oldP = image.getPixel(oldX, oldY)
            new.setPixel(newX, newY, oldP)
            oldX += factor
            newX += 1
            oldY += factor
            newY += 1
        return new
```

Reducing an image's size throws away some of its pixel information. Indeed, the greater the reduction, the greater the information loss. However, as the image becomes smaller, the human eye does not normally notice the loss of visual information, and therefore the quality of the image remains stable to perception.

The results are quite different when an image is enlarged. To increase the size of an image, you have to add pixels that were not there to begin with. In this case, you try to approximate the color values that pixels would receive if you took another sample of the subject at a higher resolution. This process can be very complex, because you also have to transform the existing pixels to blend in with the new ones that are added. Because the image gets larger, the human eye is in a better position to notice any degradation of quality when comparing it to the original. The development of a simple enlargement algorithm is left as an exercise for you.

Although we have covered only a tiny subset of the operations typically performed by an image-processing program, these operations and many more use the same underlying concepts and principles.

7.3 Exercises

- 1 Explain the advantages and disadvantages of lossless and lossy image file-compression schemes.
- 2 The size of an image is 1680 pixels by 1050 pixels. Assume that this image has been sampled using the RGB color system and placed into a raw image file. What is the minimum size of this file in megabytes? (*Hint:* There are 8 bits in a byte, 1024 bits in a kilobyte, and 1000 kilobytes in a megabyte.)
- 3 Describe the difference between Cartesian coordinates and screen coordinates.
- 4 Describe how a row-major traversal visits every position in a two-dimensional grid.
- 5 How would a column-major traversal of a grid work? Write a code segment that prints the positions visited by a column-major traversal of a 2 by 3 grid.
- 6 Explain why one would use the **clone** method with a given object.
- 7 Why does the **blur** function need to work with a copy of the original image?

Summary

- Object-based programming uses classes, objects, and methods to solve problems.
- A class specifies a set of attributes and methods for the objects of that class.
- The values of the attributes of a given object make up its state.
- A new object is obtained by instantiating its class. An object's attributes receive their initial values during instantiation.
- The behavior of an object depends on its current state and on the methods that manipulate this state.
- The set of a class's methods is called its interface. The interface is what a programmer needs to know to use objects of a class. The information in an interface usually includes the method headers and documentation about arguments, return values, and changes of state.
- Turtle graphics is a lightweight toolkit used to draw pictures in a Cartesian coordinate system. In this system, the **Turtle** object has a position, a color, a line width, a direction, and a state of being down or up with respect to a drawing window. The values of these attributes are used and changed when the **Turtle** object's methods are called.
- The RGB system represents a color value by mixing integer components that represent red, green, and blue intensities. There are 256 different values for each component, ranging from 0, indicating absence, to 255, indicating complete saturation. There are 2^{24} different combinations of RGB components for 16,777,216 unique colors.
- A grayscale system uses 8, 16, or 256 distinct shades of gray.
- Digital images are captured by sampling analog information from a light source, using a device such as a digital camera or a flatbed scanner. Each sampled color value is mapped to a discrete color value among those supported by the given color system.
- Digital images can be stored in several file formats. A raw image format preserves all of the sampled color information, but occupies the most storage space. The JPEG format uses various data-compression schemes to reduce the file size, while preserving fidelity to the original samples. Lossless schemes either preserve or reconstitute the

original samples upon decompression. Lossy schemes lose some of the original sample information. The GIF format is a lossy scheme that uses a palette of up to 256 colors and stores the color information for the image as indexes into this palette.

- During the display of an image file, each color value is mapped onto a pixel in a two-dimensional grid. The positions in this grid correspond to the screen coordinate system, in which the upper-left corner is at (0, 0), and the lower-right corner is at (*width* - 1, *height* - 1).
- A nested loop structure is used to visit each position in a two-dimensional grid. In a row-major traversal, the outer loop of this structure moves down the rows using the y-coordinate, and the inner loop moves across the columns using the x-coordinate. Each column in a row is visited before moving to the next row. A column-major traversal reverses these settings.
- Image-manipulation algorithms either transform pixels at given positions or create a new image using the pixel information of a source image. Examples of the former type of operation are conversion to black and white and conversion to grayscale. Blurring, edge detection, and altering the image size are examples of the second type of operation.

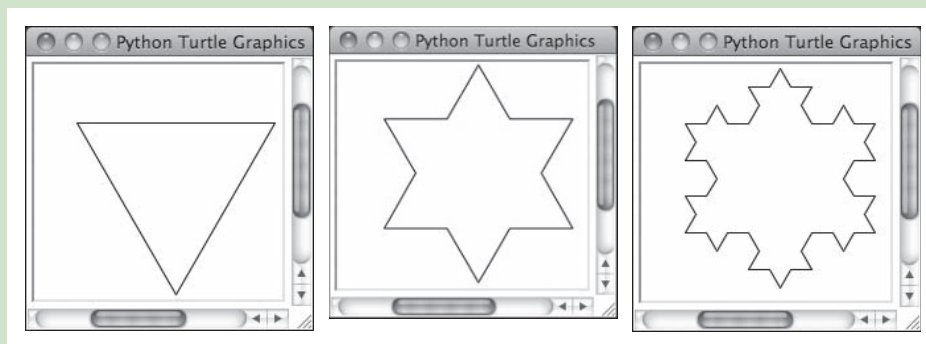
REVIEW QUESTIONS

- 1 The interface of a class is the set of all its
 - a objects
 - b attributes
 - c methods
- 2 The state of an object consists of
 - a its class of origin
 - b the values of all of its attributes
 - c its physical structure
- 3 Instantiation is a process that
 - a compares two objects for equality
 - b builds a string representation of an object
 - c creates a new object of a given class

- 4 The **print** function
 - a creates a new object
 - b copies an existing object
 - c prints a string representation of an object
- 5 The **clone** method
 - a creates a new object
 - b copies an existing object
 - c returns a string representation of an object
- 6 The origin (0, 0) in a screen coordinate system is at
 - a the center of a window
 - b the upper-left corner of a window
- 7 A row-major traversal of a two-dimensional grid visits all of the positions in a
 - a row before moving to the next row
 - b column before moving to the next column
- 8 In a system of 256 unique colors, the number of bits needed to represent each color is
 - a 4
 - b 8
 - c 16
- 9 In the RGB system, where each color contains three components with 256 possible values each, the number of bits needed to represent each color is
 - a 8
 - b 24
 - c 256
- 10 The process whereby analog information is converted to digital information is called
 - a recording
 - b sampling
 - c filtering
 - d compressing

PROJECTS

- 1 Define a function **drawCircle**. This function should expect a **Turtle** object, the coordinates of the circle's center point, and the circle's radius as arguments. The function should draw the specified circle. The algorithm should draw the circle's circumference by turning 3 degrees and moving a given distance 120 times. Calculate the distance moved with the formula $2.0 * \pi * radius / 120.0$.
- 2 Modify this chapter's case study program (the c-curve) so that it draws the line segments using random colors.
- 3 The *Koch snowflake* is a fractal shape. At level 0, the shape is an equilateral triangle. At level 1, each line segment is split into four equal parts, producing an equilateral bump in the middle of each segment. Figure 7.15 shows these shapes at levels 0, 1, and 2.

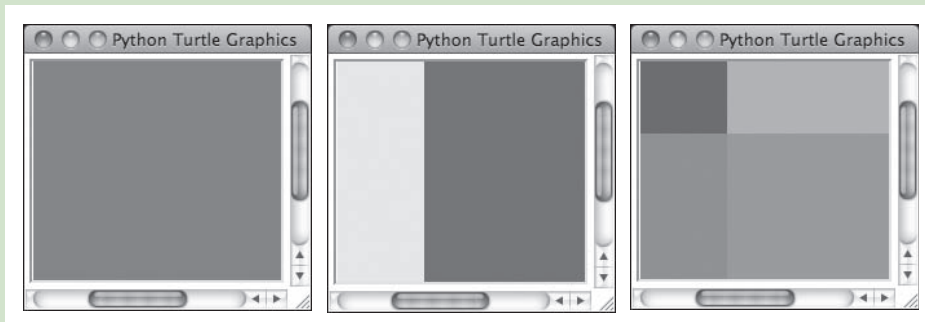


[FIGURE 7.15] First three levels of a Koch snowflake

At the top level, the script uses a function **drawFractalLine** to draw three fractal lines. Each line is specified by a given distance, direction (angle), and level. The initial angles are 0, -120, and 120 degrees. The initial distance can be any size, such as 200 pixels. The function **drawFractalLine** is recursive. If the level is 0, then the turtle moves the given distance in the given direction. Otherwise, the function draws four fractal lines with $\frac{1}{3}$ of the given distance, angles that produce the given effect, and the given level minus 1. Write a script that draws the Koch snowflake.

- 4 The twentieth century Dutch artist Piet Mondrian developed a style of abstract painting that exhibited simple recursive patterns. To generate such a pattern with a computer, one would begin with a filled rectangle

in a random color and then repeatedly fill two unequal subdivisions with random colors, as shown in Figure 7.16 (actual colors not shown).



[FIGURE 7.16] Generating a simple recursive pattern in the style of Piet Mondrian

As you can see, the algorithm continues the process of subdivision until an “aesthetically right moment” is reached. In this version, the algorithm divides the current rectangle into portions representing $\frac{1}{3}$ and $\frac{2}{3}$ of its area and alternates these subdivisions along the horizontal and vertical axes. Design, implement, and test a script that uses a recursive function to draw these patterns. You should use the **fillRectangle** function developed in Exercise 7.1, 8.

- 5 Define and test a function named **posterize**. This function expects an image and a tuple of RGB values as arguments. The function modifies the image like the **blackAndWhite** function, but uses the given RGB values instead of black.
- 6 Define a second version of the **grayscale** function that uses the allegedly crude method of simply averaging each RGB value. Test the function by comparing its results with those of the other version discussed in this chapter.
- 7 Inverting an image makes it look like a photographic negative. Define and test a function named **invert**. This function expects an image as an argument and resets each RGB component to 255 minus that component. Be sure to test the function with images that have been converted to grayscale and black and white as well as color images.
- 8 Old-fashioned photographs from the nineteenth century are not quite black and white and not quite color, but seem to have shades of gray, brown, and blue. This effect is known as **sepia**. Write and test a function named **sepia** that converts a color image to sepia. This function should first call **grayscale** to convert the color image to grayscale. A code

segment for transforming the grayscale values to achieve a sepia effect follows. Note that the value for green does not change.

```
if red < 63:
    red = int(red * 1.1)
    blue = int(blue * 0.9)
elif red < 192:
    red = int(red * 1.15)
    blue = int(blue * 0.85)
else:
    red = min(int(red * 1.08), 255)
    blue = int(blue * 0.93)
```

- 9 Darkening an image requires adjusting all of its pixels toward black as a limit, whereas lightening an image requires adjusting them toward white as a limit. Because black is RGB (0, 0, 0) and white is RGB (255, 255, 255), adjusting the three RGB values of each pixel by the same amount in either direction will have the desired effect. Of course, the algorithms have to avoid exceeding either limit during the adjustments.

Lightening and darkening are actually special cases of a process known as **color filtering**. A color filter is any RGB triple applied to an entire image. The filtering algorithm adjusts each pixel by the amounts specified in the triple. For example, you can increase the amount of red in an image by applying a color filter with a positive red value and green and blue values of 0. The filter (20, 0, 0) would make an image's overall color slightly redder. Alternatively, you can reduce the amount of red by applying a color filter with a negative red value. Once again, the algorithms have to avoid exceeding the limits on the RGB values.

Develop three algorithms for lightening, darkening, and color filtering as three related Python functions, **lighten**, **darken**, and **colorFilter**. The first two functions should expect an image and a positive integer as arguments. The third function should expect an image and a tuple of integers (the RGB values) as arguments. The following session shows how these functions can be used with the images **image1**, **image2**, and **image3**, which are initially white:

```
>>> darken(image1, 128)                # Converts to gray
>>> darken(image2, 64)                  # Converts to dark gray
>>> colorFilter(image3, (255, 0, 0))    # Converts to red
```

Note that the function **colorFilter** should do most of the work.

- 10 The edge-detection function described in this chapter returns a black-and-white image. Think of a similar way to transform color values so that the new image is still in its original colors but the outlines within it are merely sharpened. Then, define a function named **sharpen** that performs this operation. The function should expect an image and two integers as arguments. One integer should represent the degree to which the image should be sharpened. The other integer should represent the threshold used to detect edges. (*Hint*: A pixel can be darkened by making its RGB values smaller.)
- 11 To enlarge an image, one must fill in new rows and columns with color information based on the colors of neighboring positions in the original image. Develop and test a function named **enlarge**. This function should expect an image and an integer factor as arguments. The function should build and return a new image that represents the expansion of the original image by the factor. (*Hint*: Copy each row of pixels in the original image to one or more rows in the new image. To copy a row, use two index variables, one that starts on the left of the row and one that starts on the right. These two indexes converge to the middle. This will allow you to copy each pixel to one or more positions of a row in the new image.)
- 12 Each image-processing function that modifies its image argument has the same loop pattern for traversing the image. The only thing that varies is the code used to change each pixel within the loop. Section 6.6 of this book, on higher-order functions, suggests a simpler design pattern for such code. Design a single function, named **transform**, which expects an image and a function as arguments. When this function is called, it should be passed another function that expects a tuple of integers and returns a tuple of integers. This is the function that transforms the information for an individual pixel (such as converting it to black and white or grayscale). The **transform** function contains the loop logic for traversing its image argument. In the body of the loop, the **transform** function accesses the pixel at the current position, passes it as an argument to the other function, and resets the pixel in the image to the function's value. Write and test a script that defines this function and uses it to perform at least two different types of transformation on an image.

[CHAPTER] 8

Design with Classes

After completing this chapter, you will be able to:

- Determine the attributes and behavior of a class of objects required by a program
- List the methods, including their parameters and return types, that realize the behavior of a class of objects
- Choose the appropriate data structures to represent the attributes of a class of objects
- Define a constructor, instance variables, and methods for a class of objects
- Recognize the need for a class variable and define it
- Define a method that returns the string representation of an object
- Define methods for object equality and comparisons
- Exploit inheritance and polymorphism when developing classes
- Transfer objects to and from files

This book has covered the use of many software tools in computational problem solving. The most important of these tools are the abstraction mechanisms for simplifying designs and controlling the complexity of solutions. Abstraction mechanisms include functions, modules, objects, and classes. In each case, we have begun with an external view of a resource, showing what it does and how it can be used. For example, to use a function in the built-in **math** module, you import it, run **help** to learn how to use the function correctly, and then include it appropriately in your code. You follow the same procedures for built-in data structures such as strings and lists, and for library resources such as the **Turtle** and **Image** classes covered in Chapter 7. From a user's perspective, you shouldn't be concerned with how a resource performs its task. The beauty and utility of an abstraction is that it frees you from the need to be concerned with such details.

Unfortunately, not all useful abstractions are built in. You will sometimes need to custom design an abstraction to suit the needs of a specialized application or suite of applications you are developing. When designing your own abstraction, you must take a different view from that of users and concern yourself with the inner workings of a resource. The programmer who defines a new function or constructs a new module of resources is using the resources provided by others to build new software components. In this chapter, we take an internal view of objects and classes, showing how to design, implement, and test another useful abstraction mechanism—a class.

Programming languages that allow the programmer to define new classes of objects are called **object-oriented languages**. These languages also support a style of programming called **object-oriented programming**. Unlike object-based programming, which simply uses ready-made objects and classes within a framework of functions and algorithmic code, object-oriented programming sustains an effort to conceive and build entire software systems from cooperating classes. We begin this chapter by exploring the definitions of a few classes. We then discuss how cooperating classes can be organized into complex software systems. This strategy is rather different from the strategy of procedural design with functions discussed in Chapter 6. The advantages and disadvantages of each design strategy will become clear as we proceed.

8.1 Getting Inside Objects and Classes

Programmers who use objects and classes know several things:

- The interface or set of methods that can be used with a class of objects
- The attributes of an object that describe its state from the user's point of view
- How to instantiate a class to obtain an object

Like functions, objects are abstractions. A function packages an algorithm in a single operation that can be called by name. An object packages a set of data values—its state—and a set of operations—its methods—in a single entity that can be referenced with a name. This makes an object a more complex abstraction than a function. To get inside a function, you must view the code contained in its definition. To get inside an object, you must view the code contained in its class. A class definition is like a blueprint for each of the objects of that class. This blueprint contains

- Definitions of all of the methods that its objects recognize
- Descriptions of the data structures used to maintain the state of an object, or its attributes, from the implementer's point of view

To illustrate these ideas, we now present a simple class definition for a course-management application, followed by a discussion of the basic concepts involved.

8.1.1 A First Example: The **Student** Class

A course-management application needs to represent information about students in a course. Each student has a name and a list of test scores. We can use these as the attributes of a class named **Student**. The **Student** class should allow the user to view a student's name, view a test score at a given position (counting from 1), reset a test score at a given position, view the highest test score, view the average test score, and obtain a string representation of the student's information. When a **Student** object is created, the user supplies the student's name and the number of test scores. Each score is initially presumed to be 0.

The interface or set of methods of the **Student** class is described in Table 8.1. Assuming that the **Student** class is defined in a file named **student.py**, the next session shows how it could be used:

```
>>> from student import Student
>>> s = Student("Maria", 5)
>>> print(s)
Name: Maria
Scores: 0 0 0 0 0
>>> s.setScore(1, 100)
>>> print(s)
Name: Maria
Scores: 100 0 0 0 0
>>> s.getHighScore()
100
>>> s.getAverage()
20
>>> s.getScore(1)
100
>>> s.getName()
'Maria'
>>>
```

Student METHOD	WHAT IT DOES
s = Student(name, number)	Returns a Student object with the given name and number of scores. Each score is initially 0.
s.getName()	Returns the student's name.
s.getScore(i)	Returns the student's i th score. i must range from 1 through the number of scores.
s.setScore(i, score)	Resets the student's i th score to score . i must range from 1 through the number of scores.
s.getAverage()	Returns the student's average score.
s.getHighScore()	Returns the student's highest score.
s.__str__()	Same as str(s) . Returns a string representation of the student's information.

[TABLE 8.1] The interface of the **Student** class

The syntax of a simple class definition is the following:

```
class <class name>(<parent class name>):
    <method definition-1>
    ...
    <method definition-n>
```

The class definition syntax has two parts: a class header and a set of method definitions that follow the class header. The class header consists of the class name and the parent class name.

The class name is a Python identifier. Although built-in type names are not capitalized, Python programmers typically capitalize their own class names to distinguish them from variable names.

The parent class name refers to another class. All Python classes, including the built-in ones, are organized in a tree-like **class hierarchy**. At the top, or root, of this tree is the most abstract class, named **object**, which is built in. Each class immediately below another class in the hierarchy is referred to as a **subclass**, whereas the class immediately above it, if there is one, is called its **parent class**. If the parenthesized parent class name is omitted from the class definition, the new class is automatically made a subclass of **object**. In the example class definitions shown in this book, we explicitly include the parent class names. More will be said about the relationships among classes in the hierarchy later in this chapter.

The code for the **Student** class follows, and its structure is explained in the next few subsections:

```
"""
File: student.py
Resources to manage a student's name and test scores.
"""

class Student(object):
    """Represents a student."""

    def __init__(self, name, number):
        """Constructor creates a Student with the given name
        and number of scores and sets all scores to 0."""
        self._name = name
        self._scores = []
        for count in range(number):
            self._scores.append(0)

    def getName(self):
        """Returns the student's name."""
        return self._name

    def setScore(self, i, score):
        """Resets the ith score, counting from 1."""
        self._scores[i - 1] = score

    def getScore(self, i):
        """Returns the ith score, counting from 1."""
        return self._scores[i - 1]

    def getAverage(self):
        """Returns the average score."""
        return sum(self._scores) / len(self._scores)

    def getHighScore(self):
        """Returns the highest score."""
        return max(self._scores)

    def __str__(self):
        """Returns the string representation of the student."""
        return "Name: " + self._name + "\nScores: " + \
            " ".join(map(str, self._scores))
```

8.1.2 Docstrings

The first thing to note is the positioning of the docstrings in our code. They can occur at three levels. The first level is that of the module. Its purpose should be familiar to you by now. The second level is just after the class header. Because there might be more than one class defined in a module, each class can have a docstring that describes its purpose. The third level is located after each method header. Docstrings at this level serve the same role as they do for function definitions. When you enter `help(Student)` at a shell prompt, the interpreter prints the documentation for the class and all of its methods.

8.1.3 Method Definitions

All of the method definitions are indented below the class header. Because methods are a bit like functions, the syntax of their definitions is similar. Note, however, that each method definition must include a first parameter named **self**, even if that method seems to expect no arguments when called. When a method is called with an object, the interpreter binds the parameter **self** to that object so that the method's code can refer to the object by name. Thus, for example, the code

```
s.getScore(4)
```

binds the parameter **self** in the method **getScore** to the **Student** object referenced by the variable **s**. The code for **getScore** can then use **self** to access that particular object's test scores.

Otherwise, methods behave just like functions. They can have required and/or optional arguments, and they can return values. They can create and use temporary variables. A method automatically returns the value **None** when it includes no **return** statement.

8.1.4 The `__init__` Method and Instance Variables

Most classes include a special method named `__init__`. Here is the code for this method in the **Student** class:

```
def __init__(self, name, number):  
    """All scores are initially 0."""  
    self._name = name  
    self._scores = []  
    for count in range(number):  
        self._scores.append(0)
```

Note that `__init__` must begin and end with two consecutive underscores. This method is also called the class's **constructor**, because it is run automatically when a user instantiates the class. Thus, when the code segment

```
s = Student("Juan", 5)
```

is run, Python automatically runs the constructor or `__init__` method of the **Student** class. The purpose of the constructor is to initialize an individual object's attributes. In addition to **self**, the **Student** constructor expects two arguments that provide the initial values for these attributes. From this point on, when we refer to a class's constructor, we mean its `__init__` method.

The attributes of an object are represented as **instance variables**. Each individual object has its own set of instance variables. These variables serve as storage for its state. The scope of an instance variable (including **self**) is the entire class definition. Thus, all of the class's methods are in a position to reference the instance variables. The lifetime of an instance variable is the lifetime of the enclosing object. An object's lifetime will be discussed in more detail later in this chapter.

Within the class definition, the names of instance variables must begin with **self**. In this code, the instance variables **self._name** and **self._scores** are initialized to a string and a list, respectively.

Python programmers are encouraged to begin the part of an instance variable's name following the dot with a single underscore, as in **self._name**. They can use this convention to distinguish instance variable names from those of temporary variables. For example, if we had used the statement **scores = []** to initialize the list of test scores, the Python interpreter would have created a temporary variable within the constructor rather than an instance variable. The

storage for this variable would be discarded at the end of the method, leaving the new **Student** object with no instance variable for its test scores.

8.1.5 The `__str__` Method

Many built-in Python classes usually include an `__str__` method. This method builds and returns a string representation of an object's state. When the `str` function is called with an object, that object's `__str__` method is automatically invoked to obtain the string that `str` returns. For example, the function call `str(s)` is equivalent to the method call `s.__str__()`, and is simpler to write. The function call `print(s)` also automatically runs `str(s)` to obtain the object's string representation for output. Here is the code for the `__str__` method in the **Student** class:

```
def __str__(self):
    """Returns the string representation of the student."""
    return "Name: " + self._name + "\nScores: " + \
        " ".join(map(str, self._scores))
```

The programmer can return any information that would be relevant to the users of a class. Perhaps the most important use of `__str__` is in debugging, when you often need to observe the state of an object after running another method.

8.1.6 Accessors and Mutators

Methods that allow a user to observe but not change the state of an object are called **accessors**. Methods that allow a user to modify an object's state are called **mutators**. The **Student** class has just one mutator method. It allows the user to reset a test score at a given position. The remaining methods are accessors. Here is the code for the mutator method `setScore`:

```
def setScore(self, i, score):
    """Resets the ith score, counting from 1."""
    self._scores[i - 1] = score
```

In general, the fewer the number of changes that can occur to an object, the easier it is to use it correctly. That is one reason Python strings are immutable. In the case of the **Student** class, if there is no need to modify an attribute, such as a student's name, we do not include a method to do that.

8.1.7 The Lifetime of Objects

Earlier, we said that the lifetime of an object's instance variables is the lifetime of that object. What determines the span of an object's life? We know that an object comes into being when its class is instantiated. When does an object die? In Python, an object becomes a candidate for the graveyard when it can no longer be referenced by the program that created it. For example, the next session creates two references to the same **Student** object:

```
>>> s = Student("Sam", 10)
>>> csc1111 = [s]
>>> csc1111
[<__main__.Student instance at 0x11ba2b0>]
>>> s
<__main__.Student instance at 0x11ba2b0>
>>>
```

As long as one of these references survives, the **Student** object can remain alive. Continuing this session, we now sever both of these references to the **Student** object:

```
>>> s = None
>>> csc1111.pop()
<__main__.Student instance at 0x11ba2b0>
>>> print(s)
None
>>> csc1111
[]
>>>
```

The **Student** object still exists, but the interpreter will eventually recycle its storage during a process called **garbage collection**. For all intents and purposes, this object has expired, and its storage will eventually be used to create other objects.

Rules of Thumb for Defining a Simple Class

We conclude this section by listing several rules of thumb for designing and implementing a simple class:

- 1 Before writing a line of code, think about the behavior and attributes of the objects of the new class. What actions does an object perform, and how, from the external perspective of a user, do these actions access or modify the object's state?
- 2 Choose an appropriate class name, and develop a short list of the methods available to users. This interface should include appropriate method names and parameter names, as well as brief descriptions of what the methods do. Avoid describing how the methods perform their tasks.
- 3 Write a short script that appears to use the new class in an appropriate way. The script should instantiate the class and run all of its methods. Of course you will not be able to execute this script until you have completed the next few steps, but it will help to clarify the interface of your class and serve as an initial test bed for it.
- 4 Choose the appropriate data structures to represent the attributes of the class. These will be either built-in types such as integers, strings, and lists, or other programmer-defined classes.
- 5 Fill in the class template with a constructor (`__init__` method) and an `__str__` method. Remember that the constructor initializes an object's instance variables, whereas `__str__` builds a string from this information. As soon as you have defined these two methods, you can test your class by instantiating it and printing the resulting object.
- 6 Complete and test the remaining methods incrementally, working in a bottom-up manner. If one method depends on another, complete the second method first.
- 7 Remember to document your code. Include a docstring for the module, the class, and each method. Do not add these as an afterthought. Write them as soon as you write a class header or a method header. Be sure to examine the results by running **help** with the class name.

8.1 Exercises

- 1 What are instance variables, and what role does the name **self** play in the context of a class definition?
- 2 Explain what a constructor does.
- 3 The **Student** class has no mutator method that allows a user to change a student's name. Define a method **setName** that allows a user to change a student's name.
- 4 The method **getAge** expects no arguments and returns the value of an instance variable named **_age**. Write the code for the definition of this method.
- 5 How is the lifetime of an object determined? What happens to an object when it dies?

8.2 Case Study: Playing the Game of Craps

College students are known to study hard and play hard. In this case study, we develop some classes that cooperate to allow students to play and study the behavior of the game of craps.

8.2.1 Request

Write a program that allows the user to play and study the game of craps.

8.2.2 Analysis

A player in the game of craps rolls a pair of dice. If the sum of the values on this initial roll is 2, 3, or 12, the player loses. If the sum is 7 or 11, the player wins. Otherwise, the player continues to roll until the sum is 7, indicating a loss, or the sum equals the initial sum, indicating a win.

During analysis, you decide which classes of objects will be used to model the behavior of the objects in the problem domain. The classes often become evident when you consider the nouns used in the problem description. In this case, the two most significant nouns in our description of a game of craps are “player” and

“dice.” Thus, the classes will be named **Player** and **Die** (the singular, as a player will use two instances).

Analysis also specifies the roles and responsibilities of each class. You can describe these in terms of the behavior of each object in the program. A **Die** object can be rolled and its value examined. That’s about it. A **Player** object can play a complete game of craps. During the course of this game, the player keeps track of the rolls of the dice. After a game is over, the player can be asked for a history of the rolls and for the game’s outcome. The player can then play another game, and so on.

The user interface for this program prompts the user for the number of games to play. The program plays that number of games and generates and displays statistics about the results for that round of games. These results, our “study” of the game, include the number of wins, losses, rolls per win, rolls per loss, and winning percentage, for the given number of games played.

Here is a sample session with the program:

```
>>> playOneGame()  
(2, 2) 4  
(2, 1) 3  
(4, 6) 10  
(6, 5) 11  
(4, 1) 5  
(5, 6) 11  
(3, 5) 8  
(3, 1) 4  
  
You win!  
>>> playManyGames()  
Enter the number of games: 100  
The total number of wins is 49  
The total number of losses is 51  
The average number of rolls per win is 3.37  
The average number of rolls per loss is 4.20  
The winning percentage is 0.490  
>>>
```

8.2.3 Design

During design, you choose the appropriate data structures for the instance variables of each class and develop its methods using pseudocode, if necessary. You can work from class interfaces provided by analysis or develop the interfaces as

the first step of design. The interfaces of the **Die** and **Player** classes are listed in Table 8.2.

Player METHOD	WHAT IT DOES
p = Player()	Returns a new player object.
p.play()	Plays the game and returns True if there is a win, False otherwise.
p.getNumberOfRolls()	Returns the number of rolls.
p.__str__()	Same as str(p) . Returns a formatted string representation of the rolls.
Die METHOD	WHAT IT DOES
d = Die()	Returns a new die object whose initial value is 1.
d.roll()	Resets the die's value to a random number between 1 and 6.
d.getValue()	Returns the die's value.
d.__str__()	Same as str(d) . Returns the string representation of the die's value.

[TABLE 8.2] The interfaces of the **Die** and **Player** classes

A **Die** object has a single attribute, an integer ranging in value from 1 through 6. At instantiation, the instance variable **self._value** is initialized to 1. The method **roll** modifies this value by resetting it to a random number from 1 to 6. The method **getValue** returns this value. The method **__str__** returns its string representation. The **Die** class can be coded immediately without further design work.

A **Player** object has three attributes, a pair of dice and a history of rolls in its most recent game. We represent each roll as a tuple of two integers and the set of rolls as a list of these tuples. At instantiation, the instance variable **self._rolls** is set to an empty list.

The method **__str__** converts the list of rolls to a formatted string that contains a roll and the sum from that roll on each line.

The **play** method implements the logic of playing a game and tracking its results. Here is the pseudocode:

```
Create a new list of rolls
Roll the dice and add their values to the rolls list
If sum of the initial roll is 2, 3, or 12, return false
If the sum of the initial roll is 7 or 11, return true
While true
    Roll the dice and add their values to the rolls list
    If the sum of the roll is 7, return false
    Else if the sum of the roll equals the initial sum, return true
```

Note that the rolls list, which is an instance variable, is reset to an empty list on each play. That allows the same player to play multiple games.

The script that defines the **Player** and **Die** classes also includes two functions. The role of these functions is to interact with the human user by receiving inputs, playing the games, and displaying their results. The **playManyGames** function prompts the user for the number of games, creates a single **Player** object, plays the games and gathers data on the results, processes these data, and displays the required information. We also include a simpler function **playOneGame** that plays just one game and displays the results.

8.2.4 Implementation (Coding)

The **Die** class is defined in a file named **die.py**. The **Player** class and the top-level functions are defined in a file named **craps.py**. Here is the code for the two modules:

```
"""
File: die.py

This module defines the Die class.
"""

from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        """The initial face of the die."""
        self._value = 1
```

continued

```

def roll(self):
    """Resets the die's value to a random number
    between 1 and 6."""
    self._value = randint(1, 6)

def getValue(self):
    return self._value

def __str__(self):
    return str(self._value)

"""
File: craps.py

This module studies and plays the game of craps.
"""

from die import Die

class Player(object):

    def __init__(self):
        """Has a pair of dice and an empty rolls list."""
        self._die1 = Die()
        self._die2 = Die()
        self._rolls = []

    def __str__(self):
        """Returns the string rep of the history of rolls."""
        result = ""
        for (v1, v2) in self._rolls:
            result = result + str((v1, v2)) + " " + \
                str(v1 + v2) + "\n"
        return result

    def getNumberOfRolls(self):
        """Returns the number of the rolls in one game."""
        return len(self._rolls)

    def play(self):
        """Plays a game, saves the rolls for that game,
        and returns True for a win and False for a loss."""
        self._rolls = []
        self._die1.roll()
        self._die2.roll()
        (v1, v2) = (self._die1.getValue(),
                    self._die2.getValue())

```

continued

```

        self._rolls.append((v1, v2))
        initialSum = v1 + v2
        if initialSum in (2, 3, 12):
            return False
        elif initialSum in (7, 11):
            return True
        while True:
            self._die1.roll()
            self._die2.roll()
            (v1, v2) = (self._die1.getValue(),
                        self._die2.getValue())
            self._rolls.append((v1, v2))
            sum = v1 + v2
            if sum == 7:
                return False
            elif sum == initialSum:
                return True

# Functions that interact with the user to play the games

def playOneGame():
    """Plays a single game and prints the results."""
    player = Player()
    youWin = player.play()
    print(player)
    if youWin:
        print("You win!")
    else:
        print("You lose!")

def playManyGames():
    """Plays a number of games and prints statistics."""
    number = int(input("Enter the number of games: "))
    wins = 0
    losses = 0
    winRolls = 0
    lossRolls = 0
    player = Player()
    for count in range(number):
        hasWon = player.play()
        rolls = player.getNumberOfRolls()
        if hasWon:
            wins += 1
            winRolls += rolls
        else:
            losses += 1
            lossRolls += rolls

```

continued

```

print("The total number of wins is", wins)
print("The total number of losses is", losses)
print("The average number of rolls per win is %0.2f" % \
      (winRolls / wins))
print("The average number of rolls per loss is %0.2f" % \
      (lossRolls / losses))
print("The winning percentage is %0.3f" % \
      (wins / number))

```

8.3 Data-Modeling Examples

As you have seen, objects and classes are useful for modeling objects in the real world. In this section, we explore several other examples.

8.3.1 Rational Numbers

We begin with numbers. A **rational number** consists of two integer parts, a numerator and a denominator, and is written using the format *numerator / denominator*. Examples are 1/2, 1/3, and so forth. Operations on rational numbers include arithmetic and comparisons. Python has no built-in type for rational numbers. Let us develop a new class named **Rational** to support this type of data.

The interface of the **Rational** class includes a constructor for creating a rational number, an **str** function for obtaining a string representation, and accessors for the numerator and denominator. We will also show how to include methods for arithmetic and comparisons. Here is a sample session to illustrate the use of the new class:

```

>>> oneHalf = Rational(1, 2)
>>> oneSixth = Rational(1, 6)
>>> print(oneHalf)
1/2
>>> print(oneHalf + oneSixth)
2/3
>>> oneHalf == oneSixth
False
>>> oneHalf > oneSixth
True

```

Note that this session uses the built-in operators `+`, `==`, and `<` with objects of the new class, **Rational**. Python allows the programmer to **overload** many of the built-in operators for use with new data types.

We develop this class in two steps. First, we take care of the internal representation of a rational number and also its string representation. The constructor expects the numerator and denominator as arguments and sets two instance variables to this information. This method then reduces the rational number to its lowest terms. To reduce a rational number to its lowest terms, you first compute the greatest common divisor (GCD) of the numerator and the denominator, using Euclid's algorithm, as described in Programming Project 8 of Chapter 3. You then divide the numerator and the denominator by this GCD. These tasks are assigned to two other **Rational** methods, `_reduce` and `_gcd`. Because these methods are not intended to be in the class's interface, their names begin with the `_` symbol. Performing the reduction step in the constructor guarantees that it will not have to be done in any other operation. Here is the code for the first step:

```
"""
File: rational.py
Resources to manipulate rational numbers.
"""

class Rational(object):
    """Represents a rational number."""

    def __init__(self, numer, denom):
        """Constructor creates a number with the given numerator
        and denominator and reduces it to lowest terms."""
        self._numer = numer
        self._denom = denom
        self._reduce()

    def numerator(self):
        """Returns the numerator."""
        return self._numer

    def denominator(self):
        """Returns the denominator."""
        return self._denom

    def __str__(self):
        """Returns the string representation of the number."""
        return str(self._numer) + "/" + str(self._denom)
```

continued

```

def _reduce(self):
    """Helper to reduce the number to lowest terms."""
    divisor = self._gcd(self._numer, self._denom)
    self._numer = self._numer // divisor
    self._denom = self._denom // divisor

def _gcd(self, a, b):
    """Euclid's algorithm for greatest common divisor."""
    (a, b) = (max(a, b), min(a, b))
    while b > 0:
        (a, b) = (b, a % b)
    return a

# Methods for arithmetic and comparisons go here

```

You can now test the class by instantiating numbers and printing them. When you are satisfied that the data are being represented correctly, you can move on to the next step.

8.3.2 Rational Number Arithmetic and Operator Overloading

We now add methods to perform arithmetic with rational numbers. Recall that the earlier session used the built-in operators for arithmetic. For a built-in type such as **int** or **float**, each arithmetic operator corresponds to a special method name. You will see many of these methods by entering **dir(int)** or **dir(str)** at a shell prompt, and they are listed in Table 8.3. The object on which the method is called corresponds to the left operand, whereas the method's second parameter corresponds to the right operand. Thus, for example, the code **x + y** is actually shorthand for the code **x._add_(y)**.

OPERATOR	METHOD NAME
+	<code>__add__</code>
-	<code>__sub__</code>
*	<code>__mul__</code>
/	<code>__div__</code>
%	<code>__mod__</code>

[TABLE 8.3] Built-in arithmetic operators and their corresponding methods

To overload an arithmetic operator, you just define a new method using the appropriate method name. The code for each method applies a rule of rational number arithmetic. The rules are listed in Table 8.4.

TYPE OF OPERATION	RULE
Addition	$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1) / d_1d_2$
Subtraction	$n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1) / d_1d_2$
Multiplication	$n_1/d_1 * n_2/d_2 = n_1n_2 / d_1d_2$
Division	$n_1/d_1 / n_2/d_2 = n_1d_2 / d_1n_2$

[TABLE 8.4] Rules for rational number arithmetic

Each method builds and returns a new rational number that represents the result of the operation. Here is the code for the addition operation:

```
def __add__(self, other):
    """Returns the sum of the numbers."""
    #Self is the left operand and other is the right operand
    newNumer = self._numer * other._denom + \
        other._numer * self._denom
    newDenom = self._denom * other._denom
    return Rational(newNumer, newDenom)
```

Note that the parameter **self** is viewed as the left operand of the operator, whereas the parameter **other** is viewed as the right operand. The instance variables of the rational number named **other** are accessed in the same manner as the instance variables of the rational number named **self**.

Operator overloading is another example of an abstraction mechanism. In this case, programmers can use operators with single, standard meanings even though the underlying operations vary from data type to data type.

8.3.3 Comparison Methods

You can compare integers and floating-point numbers using the operators **==**, **!=**, **<**, **>**, **<=**, and **>=**. When the Python interpreter encounters one of these operators, it uses a corresponding method defined in the **float** or **int** class. Each of these methods expects two arguments. The first argument, **self**, represents the operand to the left of the operator, and the second argument represents the

other operand. Table 8.5 lists the comparison operators and the corresponding methods.

OPERATOR	MEANING	METHOD
<code>==</code>	Equals	<code>__eq__</code>
<code>!=</code>	Not equals	<code>__neq__</code>
<code><</code>	Less than	<code>__lt__</code>
<code><=</code>	Less than or equal	<code>__le__</code>
<code>></code>	Greater than	<code>__gt__</code>
<code>>=</code>	Greater than or equal	<code>__ge__</code>

[TABLE 8.5] The comparison operators and methods

To use the comparison operators with a new class of objects, such as rational numbers, the class must include these methods with the appropriate comparison logic. However, once the implementer of the class has defined methods for `==` and `<`, the remaining methods can be defined in terms of these two.

Let's implement `<` here and wait on `==` until the next section. The simplest way to compare two rational numbers is to compare the product of the extremes and the product of the means. The extremes are the first numerator and the second denominator, whereas the means are the second numerator and the first denominator. Thus, the comparison of $1/6$ and $2/3$ translates to $1 * 3 < 2 * 6$. The implementation of the `__lt__` method for rational numbers uses this strategy, as follows:

```
def __lt__(self, other):
    """Compares two rational numbers, self and other, using <."""
    extremes = self._numer * other._denom
    means = other._numer * self._denom
    return extremes < means
```

When objects of a new class are comparable, it's a good idea to include the comparison methods in that class. Then, other built-in methods, such as the **sort** method for lists, will be able to use your objects appropriately.

8.3.4 Equality and the `__eq__` Method

Equality is a different kind of relationship from the other types of comparisons. Not all objects are comparable using less than or greater than, but any two objects can be compared for equality or inequality. For example, when the variable `twoThirds` refers to a rational number, it does not make sense to say `twoThirds < "hi there"`, but it does make sense to say `twoThirds != "hi there"` (true, they aren't the same). Put another way, the first expression should generate a semantic error, whereas the second expression should return `True`.

The Python interpreter picks out equality from the other comparisons by looking for an `__eq__` method when it encounters the `==` and `!=` operators. Thus, you can include an `__eq__` method in a class to support equality tests with any types of objects. Here is the code for this method in the `Rational` class:

```
def __eq__(self, other):
    """Tests self and other for equality."""
    if self is other:                # Object identity?
        return True
    elif type(self) != type(other):  # Types match?
        return False
    else:
        return self._numer == other._numer and \
               self._denom == other._denom
```

Note that the method first tests the two operands for object identity using Python's `is` operator. The `is` operator returns `True` if `self` and `other` refer to the exact same object. If the two objects are distinct, the method then uses Python's `type` function to determine whether or not they are of the same type. If they are not of the same type, they cannot be equal. Finally, if the two operands are of the same type, the second one must be a rational number, so it is safe to access the components of both operands to compare them for equality in the last alternative.

As a rule of thumb, you should include an `__eq__` method in any class where a comparison for equality uses a criterion other than object identity, and also include the other comparison methods when the objects are comparable using less than or greater than.

8.3.5 Savings Accounts and Class Variables

Turning to the world of finance, banking systems are easily modeled with classes. For example, a savings account allows owners to make deposits and withdrawals. These accounts also compute interest periodically. A simplified version of a savings account includes an owner's name, PIN, and balance as attributes. The interface for a **SavingsAccount** class is listed in Table 8.6.

SavingsAccount METHOD	WHAT IT DOES
<code>a = SavingsAccount(name, pin, balance = 0.0)</code>	Returns a new account with the given name, PIN, and balance.
<code>a.deposit(amount)</code>	Deposits the given amount to the account's balance.
<code>a.withdraw(amount)</code>	Withdraws the given amount from the account's balance.
<code>a.getBalance()</code>	Returns the account's balance.
<code>a.getName()</code>	Returns the account's name.
<code>a.getPin()</code>	Returns the account's PIN.
<code>a.computeInterest()</code>	Computes the account's interest and deposits it.
<code>__str__(a)</code>	Same as <code>str(a)</code> . Returns the string representation of the account.

[TABLE 8.6] The interface for **SavingsAccount**

When the interest is computed, a rate is applied to the balance. If you assume that the rate is the same for all accounts, then it does not have to be maintained as an instance variable. Instead, you can use a **class variable**. A class variable is visible to all instances of a class and does not vary from instance to instance. While it normally behaves like a constant, in some situations a class variable can be modified. But when it is, the change takes effect for the entire class.

To introduce a class variable, we place the assignment statement that initializes it between the class header and the first method definition. For clarity, class variables are written in uppercase only. The code for **SavingsAccount** shows the definition and use of the class variable **RATE**. Completion of some methods is left as an exercise for you.

```

class SavingsAccount(object):
    """This class represents a Savings account
    with the owner's name, PIN, and balance."""

    RATE = 0.02

    def __init__(self, name, pin, balance = 0.0):
        self._name = name
        self._pin = pin
        self._balance = balance

    def __str__(self):
        result = 'Name:      ' + self._name + '\n'
        result += 'PIN:       ' + self._pin + '\n'
        result += 'Balance: ' + str(self._balance)
        return result

    def getBalance(self):
        return self._balance

    def getName(self):
        return self._name

    def getPin(self):
        return self._pin

    def deposit(self, amount):
        """Deposits the given amount and returns the
        new balance."""
        self._balance += amount
        return self._balance

    def withdraw(self, amount):
        """Withdraws the given amount.
        Returns None if successful, or an
        error message if unsuccessful."""
        if amount < 0:
            return 'Amount must be >= 0'
        elif self._balance < amount:
            return 'Insufficient funds'
        else:
            self._balance -= amount
            return None

    def computeInterest(self):
        """Computes, deposits, and returns the interest."""
        interest = self._balance * SavingsAccount.RATE
        self.deposit(interest)
        return interest

```

When referenced, a class variable must be preceded by the class name and a dot, as in **SavingsAccount.RATE**. Class variables are visible both inside a class definition and to external users of the class.

In general, you should use class variables only for symbolic constants or to maintain data held in common by all objects of a class. For data that are owned by individual objects, you must use instance variables instead.

8.3.6 Putting the Accounts into a Bank

Savings accounts only make sense in the context of a bank. A very simple bank allows a user to add new accounts, remove accounts, get existing accounts, and compute interest on all accounts. A **Bank** class thus has these four basic operations (**add**, **remove**, **get**, and **computeInterest**) and a constructor. This class, of course, also includes the usual **str** function for development and debugging. We assume that both **SavingsAccount** and **Bank** are defined in a file named **bank.py**. Here is a sample session that uses a **Bank** object and some **SavingsAccount** objects. The interface for **Bank** is listed in Table 8.7.

```
>>> from bank import Bank, SavingsAccount
>>> bank = Bank()
>>> bank.add(SavingsAccount("Wilma", "1001", 4000.00))
>>> bank.add(SavingsAccount("Fred", "1002", 1000.00))
>>> print(bank)
Name:      Fred
PIN:       1002
Balance: 1000.00
Name:      Wilma
PIN:       1001
Balance: 4000.00
>>> account = bank.get("1000")
>>> print(account)
None
>>> account = bank.get("1001")
>>> print(account)
Name:      Wilma
PIN:       1001
Balance: 4000.00
>>> account.deposit(25.00)
4025
>>> print(account)
Name:      Wilma
PIN:       1001
Balance: 4025.00
>>> print(bank)
```

continued

```

Name:    Fred
PIN:     1002
Balance: 1000.00
Name:    Wilma
PIN:     1001
Balance: 4025.00
>>>

```

Bank METHOD	WHAT IT DOES
b = Bank()	Returns a bank.
b.add(account)	Adds the given account to the bank.
b.remove(pin)	Removes the account with the given pin from the bank and returns the account. If the pin is not in the bank, returns None .
b.get(pin)	Returns the account associated with the pin if the PIN is in the bank. Otherwise, returns None .
b.computeInterest()	Computes the interest on each account, deposits it in that account, and returns the total interest.
__str__(b)	Same as str(b) . Returns a string representation of the bank (all the accounts).

[TABLE 8.7] The interface for the **Bank** class

To keep the design simple, the bank maintains the accounts in no particular order. Thus, you can choose a dictionary keyed by owners' PINs to represent the collection of accounts. Access and removal then depend on an owner's PIN. Here is the code for the **Bank** class:

```

class Bank(object):

    def __init__(self):
        self._accounts = {}

    def __str__(self):
        """Return the string rep of the entire bank."""
        return '\n'.join(map(str, self._accounts.values()))

    def add(self, account):
        """Inserts an account using its PIN as a key."""

```

continued

```

self._accounts[account.getPin()] = account

def remove(self, pin):
    return self._accounts.pop(pin, None)

def get(self, pin):
    return self._accounts.get(pin, None)

def computeInterest(self):
    """Computes interest for each account and
    returns the total."""
    total = 0.0
    for account in self._accounts.values():
        total += account.computeInterest()
    return total

```

Note the use of the value **None** in the methods **remove** and **get**. In this context, **None** indicates to the user that the given PIN is not in the bank.

8.3.7 Using **pickle** for Permanent Storage of Objects

Chapter 4 discussed saving data in permanent storage with text files. You can convert any object to text for storage, but the mapping of complex objects to text and back again can be tedious and cause maintenance headaches. Fortunately, Python includes a module that allows the programmer to save and load objects using a process called **pickling**. The term comes from the process of converting cucumbers to pickles for preservation in jars. However, in the case of computational objects, you can get the cucumbers back again. You can pickle an object before it is saved to a file, and then unpickle it as it is loaded from a file into a program. Python takes care of all of the conversion details automatically.

You start by importing the **pickle** module. Files are opened for input and output and closed in the usual manner, except that the flags “rb” and “wb” are used instead of “r” and “w”, respectively. To save an object, you use the function **pickle.dump**. Its first argument is the object to be “dumped,” or saved to a file, and its second argument is the file object.

You can use the **pickle** module to save the accounts in a bank to a file. You start by defining a **Bank** method named **save**. The method includes an optional argument for the filename. You assume that the **Bank** object also has an instance variable for the filename. For a new, empty bank, this variable’s value is initially **None**. Whenever the bank is saved to a file, this variable becomes the current filename. When the method’s filename argument is not provided, the method uses the bank’s current filename if there is one. This is similar to using the **Save** option in a **File** menu. When the filename argument is provided, it is

used to save the bank to a different file. This is similar to the **Save As** option in a **File** menu. Here is the code:

```
import pickle

def save(self, fileName = None):
    """Saves pickled accounts to a file. The parameter
    allows the user to change filenames."""
    if fileName != None:
        self._fileName = fileName
    elif self._fileName == None:
        return
    fileObj = open(self._fileName, 'wb')
    for account in self._accounts.values():
        pickle.dump(account, fileObj)
    fileObj.close()
```

8.3.8 Input of Objects and the **try-except** Statement

You can load pickled objects into a program from a file using the function **pickle.load**. If the end of the file has been reached, this function raises an exception. This complicates the input process, because we have no apparent way to detect the end of the file before the exception is raised. However, Python's **try-except** statement comes to our rescue. This statement allows an exception to be caught and the program to recover. The syntax of a simple **try-except** statement is the following:

```
try:
    <statements>
except <exception type>:
    <statements>
```

When this statement is run, the statements within the **try** clause are executed. If one of these statements raises an exception, control is immediately transferred to the **except** clause. If the type of exception raised matches the type in this clause, its statements are executed. Otherwise, control is transferred to the caller of the **try-except** statement and further up the chain of calls, until the exception is successfully handled or the program halts with an error message. If the statements in the **try** clause raise no exceptions, the **except** clause is skipped, and control proceeds to the end of the **try-except** statement.

We can now construct an input file loop that continues to load objects until the end of the file is encountered. When this happens, an **EOFError** is raised. The **except** clause then closes the file and breaks out of the loop. We also add a new instance variable to track the bank's filename for saving the bank to a file. Here is the code for a **Bank** method `__init__` that can take some initial accounts from an input file. This method now either creates a new, empty bank if the filename is not present, or loads accounts from a file into a **Bank** object.

```
def __init__(self, fileName = None):
    """Creates a new dictionary to hold the accounts.
    If a filename is provided, loads the accounts from
    a file of pickled accounts."""
    self._accounts = {}
    self._fileName = fileName
    if fileName != None:
        fileObj = open(fileName, 'rb')
        while True:
            try:
                account = pickle.load(fileObj)
                self.add(account)
            except EOFError:
                fileObj.close()
                break
```

8.3.9 Playing Cards

A standard deck of cards has 52 cards. There are four suits: spades, hearts, diamonds, and clubs. Each suit contains 13 cards. Each card also has a rank, which is a number used to sort the cards and determine the count in a hand. The literal numbers are 2 through 10. An ace counts as the number 1 or some other number, depending on the game being played. The face cards, jack, queen, and king, often count as 11, 12, and 13, respectively.

A **Card** class and a **Deck** class would be useful resources for game-playing programs. A **Card** object has two instance attributes, a rank and a suit. The **Card** class has two class attributes, the set of all suits and the set of all ranks. You can represent these two sets of attributes as instance variables and class variables in the **Card** class.

Because the attributes are only accessed and never modified, we do not include any methods other than a `__str__` method for the string representation. The `__init__` method expects an integer rank and a string suit as arguments and

returns a new card with that rank and suit. The next session shows the use of the **Card** class:

```
>>> threeOfSpades = Card(3, "Spades")
>>> jackOfSpades = Card(11, "Spades")
>>> print(jackOfSpades)
Jack of Spades
>>> threeOfSpades.rank < jackOfSpades.rank
True
>>> print(jackOfSpades.rank, jackOfSpades.suit)
11 Spades
```

Note that you access the rank and suit of a **Card** object by using a dot followed by the instance variable names. A card is little more than a container of two data values. Here is the code for the **Card** class:

```
class Card(object):
    """ A card object with a suit and rank."""

    RANKS = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)

    SUITS = ('Spades', 'Diamonds', 'Hearts', 'Clubs')

    def __init__(self, rank, suit):
        """Creates a card with the given rank and suit."""
        self.rank = rank
        self.suit = suit

    def __str__(self):
        """Returns the string representation of a card."""
        if self.rank == 1:
            rank = 'Ace'
        elif self.rank == 11:
            rank = 'Jack'
        elif self.rank == 12:
            rank = 'Queen'
        elif self.rank == 13:
            rank = 'King'
        else:
            rank = self.rank
        return str(rank) + ' of ' + self.suit
```

Unlike an individual card, a deck has significant behavior that can be specified in an interface. One can shuffle the deck, deal a card, and determine the number of cards left in it. Table 8.8 lists the methods of a **Deck** class and what they do. Here is a sample session that tries out a deck:

```
>>> deck = Deck()
>>> print(deck)
--- the print reps of 52 cards, in order of suit and rank
>>> deck.shuffle()
>>> len(deck)
52
>>> while len(deck) > 0:
    card = deck.deal()
    print(card)

--- the print reps of 52 randomly ordered cards
>>> len(deck)
0
```

Deck METHOD	WHAT IT DOES
d = Deck()	Returns a deck.
d.__len__()	Same as len(d) . Returns the number of cards currently in the deck.
d.shuffle()	Shuffles the cards in the deck.
d.deal()	If the deck is not empty, removes and returns the topmost card. Otherwise, returns None .
d.__str__()	Same as str(d) . Returns a string representation of the deck (all the cards in it).

[TABLE 8.8] The interface for the **Deck** class

During instantiation, all 52 unique cards are created and inserted into a deck's internal list of cards. The **Deck** constructor makes use of the class variables **RANKS** and **SUITS** in the **Card** class to order the new cards appropriately. The **shuffle** method simply passes the list of cards to **random.shuffle**. The **deal**

method removes and returns the first card in the list, if there is one, or returns the value **None** otherwise. The **len** function, like the **str** function, calls a method (in this case, **__len__**) that returns the length of the list of cards. Here is the code for **Deck**:

```
import random

# The definition of the Card class goes here

class Deck(object):
    """ A deck containing 52 cards."""

    def __init__(self):
        """Creates a full deck of cards."""
        self._cards = []
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                c = Card(rank, suit)
                self._cards.append(c)

    def shuffle(self):
        """Shuffles the cards."""
        random.shuffle(self._cards)

    def deal(self):
        """Removes and returns the top card or None
        if the deck is empty."""
        if len(self) == 0:
            return None
        else:
            return self._cards.pop(0)

    def __len__(self):
        """Returns the number of cards left in the deck."""
        return len(self._cards)

    def __str__(self):
        """Returns the string representation of a deck."""
        result = ''
        for c in self._cards:
            result = result + str(c) + '\n'
        return result
```

8.3

Exercises

- 1 Although the use of a PIN to identify a person's bank account is simple, it's not very realistic. Real banks typically assign a unique 12-digit number to each account and use this as well as the customer's PIN during a login at an ATM. Suggest how to rework the banking system discussed in this section to use this information.
- 2 What is a class variable? When should the programmer define a class variable rather than an instance variable?
- 3 Describe how the arithmetic operators can be overloaded to work with a new class of numbers.
- 4 Define a method for the **Bank** class that returns the total assets in the bank (the sum of all account balances).
- 5 Describe the benefits of pickling objects for file storage.
- 6 Why would you use a **try-except** statement in a program?
- 7 Two playing cards can be compared by rank. For example, an ace is less than a 2. When **c1** and **c2** are cards, **c1.rank < c2.rank** expresses this relationship. Explain how a method could be added to the **Card** class to simplify this expression to **c1 < c2**.

8.4

Case Study: An ATM

In this case study, we develop a simple ATM program that uses the **Bank** and **SavingsAccount** classes discussed in the previous section.

8.4.1

Request

Write a program that simulates a simple ATM.

8.4.2

Analysis

Our ATM user logs in with a name and a personal identification number, or PIN. If either string is unrecognized, Python prints an error message. Otherwise, the

user can repeatedly select options to get the balance, make a deposit, and make a withdrawal. A final option allows the user to quit. The ATM program runs until a user enters the password “CloseItDown,” so it can accept more users. Figure 8.1 shows the sample terminal-based interface.

```
ken% python atm.py
Enter your name: Name1
Enter your PIN: 1111
Error, unrecognized PIN
Enter your name: Name1
Enter your PIN: 1000
1 View your balance
2 Make a deposit
3 Make a withdrawal
4 Quit

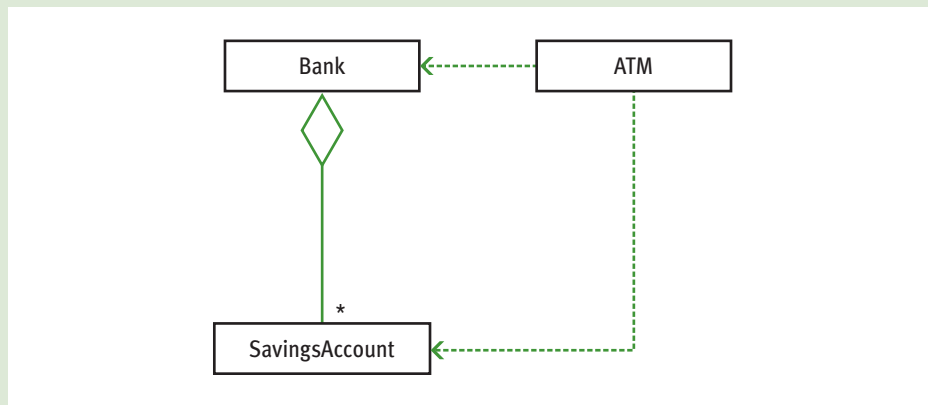
Enter a number: 1
Your balance is $ 100.0
1 View your balance
2 Make a deposit
3 Make a withdrawal
4 Quit

Enter a number: 2
Enter the amount to deposit: 50
1 View your balance
2 Make a deposit
3 Make a withdrawal
4 Quit

Enter a number: 4
Have a nice day!
Enter your name: CloseItDown
>>>
```

[FIGURE 8.1] The user interface for the ATM program

The data model classes for the program are the **Bank** and **SavingsAccount** classes developed earlier in this chapter. To support user interaction, we also develop a new class called **ATM**. The **class diagram** in Figure 8.2 shows the relationships among these classes.



[FIGURE 8.2] A UML diagram for the ATM program showing the program's classes

In a class diagram, the name of each class appears in a box. The lines or edges connecting the boxes show the relationships. Note that these edges are labeled or contain arrows. This information describes the number of accounts in a bank (zero or more) and the dependency of one class on another (the direction of an arrow). Class diagrams of this type are part of a graphical notation called the Unified Modeling Language, or UML. UML is used to describe and document the analysis and design of complex software systems.

In general, it is a good idea to divide the code for most interactive applications into at least two sets of classes. One set of classes, which we call the **view**, handles the program's interactions with human users, including the input and output operations. The other set of classes, called the **model**, represents and manages the data used by the application. In the current case study, the **Bank** and **SavingsAccount** classes belong to the model, whereas the **ATM** class belongs to the view. One of the benefits of this separation of responsibilities is that you can write different views for the same data model, such as a terminal-based view and a graphical-based view, without changing a line of code in the data model. Alternatively, you can write different representations of the data model without altering a line of code in the views. In most of the case studies that follow, we apply this framework, called the **model/view pattern**, to structure the code.

8.4.3 Design

The **ATM** class maintains two instance variables. Their values are the following:

- A **Bank** object
- The **SavingsAccount** of the currently logged-in user

At program start-up, a **Bank** object is loaded from a file. An **ATM** object is then created for this bank. The ATM's **run** method is then called. This method enters a loop that waits for a user to enter a name and a PIN. If the name equals a secret code, then the loop terminates. If the name and PIN match those of an account, the ATM's account variable is set to the user's account, and the ATM's **_processAccount** method is called. This method displays a menu of the four options. The selection of an option triggers a lower-level method to process that option. Table 8.9 lists the methods in the **ATM** class.

ATM METHOD	WHAT IT DOES
ATM(bank)	Returns a new ATM object based on bank .
run()	Starts a loop that waits for users to log in. Entering a secret code for the name terminates this process.
_processAccount()	Displays a menu of options for a logged-in user and calls the appropriate methods to handle the options.
_getBalance()	Displays the user's balance.
_deposit()	Allows the user to make a deposit.
_withdraw()	Allows the user to make a withdrawal and displays any error messages.
_quit()	Saves the bank to its file, resets the current account to None , and returns to the login loop.

[TABLE 8.9] The interface for the **ATM** class

Note that the names of all of the methods except **run** begin with the **_** symbol. The **run** method is the only method called by the user of the **ATM** class. The other methods are auxiliary methods used to accomplish tasks within the **ATM** class.

The **ATM** constructor receives a **Bank** object as an argument and saves a reference to it in an instance variable. It also sets the current account to **None** and fills a jump table, which we discussed in Chapter 6, with the lower-level methods that carry out the commands.

The **run** method logs in a user, sets the account variable, and calls **_processAccount**.

The **_processAccount** method displays a menu, inputs a user's command number, and attempts to locate a method for that number in the jump table. If a method is not found, an error message is displayed; otherwise, the method is run. If the method logs the user out, the account will equal **None**, so the command loop can break.

8.4.4 Implementation (Coding)

Before you can run this program, you need to create a bank file. We include a simple function that loads a **Bank** object with a number of dummy accounts and saves it to a file.

The code in **atm.py** defines the **ATM** class, instantiates a **Bank** and an **ATM**, and executes the ATM's **run** method. Here is the text of that file:

```
"""
File: atm.py

This module defines the ATM class and its application.

To test, launch from IDLE and run

>>> createBank(5)
>>> main()

Can be modified to run as a script after a bank has been saved.
"""

from bank import Bank, SavingsAccount

class ATM(object):
    """This class handles terminal-based ATM transactions."""

    SECRET_CODE = "CloseItDown"

    def __init__(self, bank):
        self._account = None
        self._bank = bank
        self._methods = {}          # Jump table for commands
        self._methods["1"] = self._getBalance
        self._methods["2"] = self._deposit
        self._methods["3"] = self._withdraw
        self._methods["4"] = self._quit

    def run(self):
        """Logs in users and processes their accounts."""
        while True:
            name = input("Enter your name: ")
            if name == ATM.SECRET_CODE:
                break
```

continued

```

        pin = input("Enter your PIN: ")
        self._account = self._bank.get(pin)
        if self._account == None:
            print("Error, unrecognized PIN")
        elif self._account.getName() != name:
            print("Error, unrecognized name")
            self._account = None
        else:
            self._processAccount()

def _processAccount(self):
    """A menu-driven command processor for a user."""
    while True:
        print("1  View your balance")
        print("2  Make a deposit")
        print("3  Make a withdrawal")
        print("4  Quit\n")
        number = input("Enter a number: ")
        theMethod = self._methods.get(number, None)
        if theMethod == None:
            print("Unrecognized number")
        else:
            theMethod()                # Call the method
            if self._account == None:
                break

def _getBalance(self):
    print("Your balance is $", self._account.getBalance())

def _deposit(self):
    amount = float(input("Enter the amount to deposit: "))
    self._account.deposit(amount)

def _withdraw(self):
    amount = float(input("Enter the amount to withdraw: "))
    message = self._account.withdraw(amount)
    if message:
        print(message)

def _quit(self):
    self._bank.save()
    self._account = None
    print("Have a nice day!")

# Top-level functions
def main():
    """Instantiate a Bank and an ATM and run it."""
    bank = Bank("bank.dat")

```

continued

```

atm = ATM(bank)
atm.run()

def createBank(number = 0):
    """Saves a bank with the specified number of accounts.
    Used during testing."""
    bank = Bank()
    for i in range(number):
        bank.add(SavingsAccount('Name' + str(i + 1),
                                str(1000 + i),
                                100.00))

    bank.save("bank.dat ")

```

8.5 Structuring Classes with Inheritance and Polymorphism

Object-based programming involves the use of objects, classes, and methods to solve problems. Most object-oriented languages require the programmer to master the following techniques:

- 1 **Data encapsulation.** Restricting the manipulation of an object's state by external users to a set of method calls.
- 2 **Inheritance.** Allowing a class to automatically reuse and extend the code of similar but more general classes.
- 3 **Polymorphism.** Allowing several different classes to use the same general method names.

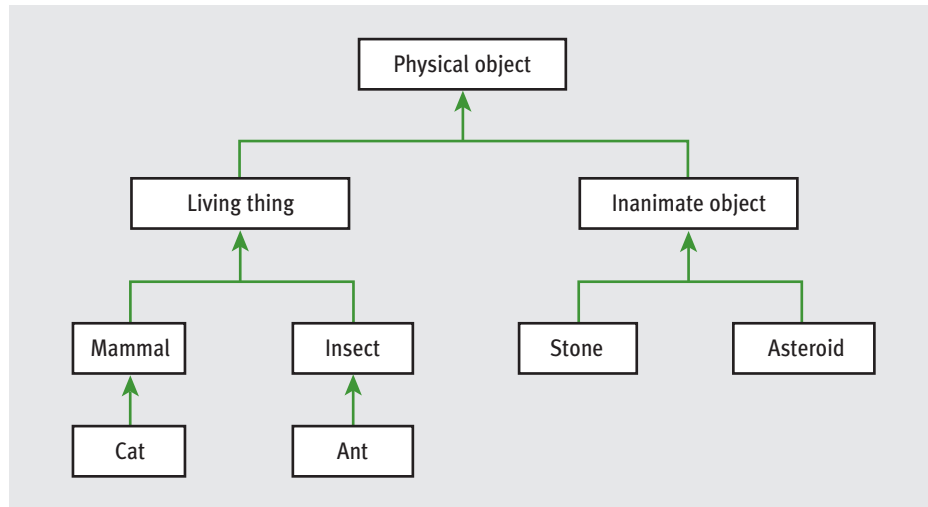
Although Python is considered an object-oriented language, its syntax does not enforce data encapsulation. However, Python programmers can adopt conventions, such as those we have used, to achieve data encapsulation in practice. For example, the use of an underscore symbol in an instance variable can dissuade an external user from writing code to access the variable in an inappropriate manner.

Unlike data encapsulation, inheritance and polymorphism are built into Python's syntax. In this section we examine how they can be exploited to structure code.

8.5.1

Inheritance Hierarchies and Modeling

Objects in the natural world and objects in the world of artifacts can be classified using **inheritance hierarchies**. A simplified hierarchy of natural objects is depicted in Figure 8.3.



[FIGURE 8.3] A simplified hierarchy of objects in the natural world

At the top of a hierarchy is the most general class of objects. This class defines features that are common to every object in the hierarchy. For example, every physical object has a mass. Classes just below this one have these features as well as additional ones. Thus, a living thing has a mass and can also grow and die. The path from a given class back up to the topmost one goes through all of that given class's ancestors. Each class below the topmost one inherits attributes and behaviors from its ancestors and extends these with additional attributes and behavior.

An object-oriented software system models this pattern of inheritance and extension in real-world systems by defining classes that extend other classes. In Python, all classes automatically extend the built-in **object** class, which is the most general class possible. However, it is possible to extend any existing class using the syntax

```
class <new class name>(<existing class name>):
```

Thus, for example, **PhysicalObject** would extend **object**, **LivingThing** would extend **PhysicalObject**, and so on.

The real advantage of inheritance in a software system is that each new subclass acquires all of the instance variables and methods of its ancestor classes for free. Like function definitions and class definitions, inheritance hierarchies provide an abstraction mechanism that allows the programmer to avoid reinventing the wheel or writing redundant code. To see how inheritance works in Python, we now explore two examples.

8.5.2 Example: A Restricted Savings Account

So far, our examples have focused on ordinary savings accounts. Banks also provide customers with restricted savings accounts. These are like ordinary savings accounts in most ways, but with some special features, such as allowing only a certain number of deposits or withdrawals a month. Let's assume that a savings account has a name, a PIN, and a balance. You can make deposits and withdrawals and access the attributes. Let's also assume that this restricted savings account permits only three withdrawals per month. The next session shows an interaction with a **RestrictedSavingsAccount** that permits up to three withdrawals:

```
>>> account = RestrictedSavingsAccount("Ken", "1001", 500.00)
>>> print(account)
Name:      Ken
PIN:       1001
Balance: 500.0
>>> account.getBalance()
500.0
>>> for count in range(3):
        account.withdraw(100)

>>> account.withdraw(50)
'No more withdrawals this month'
>>> account.resetCounter()
>>> account.withdraw(50)
```

The fourth withdrawal has no effect on the account, and it returns an error message. A new method named **resetCounter** is called to enable withdrawals for the next month.

If **RestrictedSavingsAccount** is defined as a subclass of **SavingsAccount**, every method but **withdraw** can simply be inherited and used without changes. The **withdraw** method is redefined in **RestrictedSavingsAccount** to return an error message if the number of withdrawals has exceeded the maximum. The

maximum will be maintained in a new class variable, and the monthly count of withdrawals will be tracked in a new instance variable. Finally, a new method, **resetCounter**, is included to reset the number of withdrawals to 0 at the end of each month. Here is the code for the **RestrictedSavingsAccount** class, followed by a brief explanation:

```
"""
File: savings.py

This module defines the RestrictedSavingsAccount class.
"""
from bank import SavingsAccount

class RestrictedSavingsAccount(SavingsAccount):
    """This class represents a restricted savings account."""

    MAX_WITHDRAWALS = 3

    def __init__(self, name, pin, balance = 0.0):
        """Same attributes as SavingsAccount, but with
        a counter for withdrawals."""
        SavingsAccount.__init__(self, name, pin, balance)
        self._counter = 0

    def withdraw(self, amount):
        """Restricts number of withdrawals to MAX_WITHDRAWALS."""
        if self._counter == RestrictedSavingsAccount.MAX_WITHDRAWALS:
            return "No more withdrawals this month"
        else:
            message = SavingsAccount.withdraw(self, amount)
            if message == None:
                self._counter += 1
            return message

    def resetCounter(self):
        self._counter = 0
```

The **RestrictedSavingsAccount** class includes a new class variable not found in **SavingsAccount**. This variable, called **MAX_WITHDRAWALS**, is used to restrict the number of withdrawals that are permitted per month.

The **RestrictedSavingsAccount** constructor first calls the constructor in the **SavingsAccount** class to initialize the instance variables for the name, PIN, and balance defined there. The syntax uses the class name before the dot, and explicitly includes **self** as the first argument. The general form of the syntax for

calling a method in the parent class from within a method with the same name in a subclass follows:

```
<parent class name>.<method name>(self, <other arguments>)
```

Continuing in **RestrictedSavingsAccount**'s constructor, the new instance variable **_counter** is then set to 0. The rule of thumb to remember when writing the constructor for a subclass is that each class is responsible for initializing its own instance variables. Thus, the constructor of the parent class should always be called.

The **withdraw** method is redefined in **RestrictedSavingsAccount** to override the definition of the same method in **SavingsAccount**. You allow a withdrawal only when the counter's value is less than the maximum, and you increment the counter only after a withdrawal is successful. Note that this version of the method calls the same method in the parent or superclass to perform the actual withdrawal. The syntax for this is the same as is used in the constructor.

Finally, the new method **resetCounter** is included to allow the user to continue withdrawals in the next month.

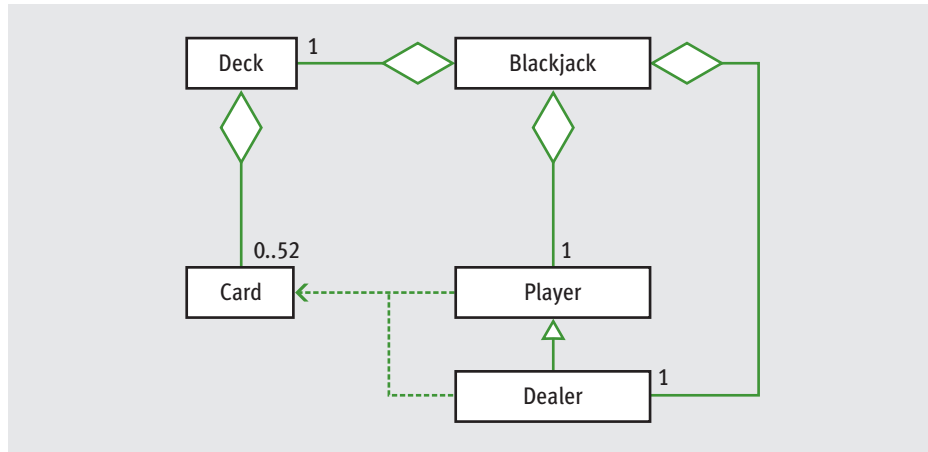
8.5.3

Example: The Dealer and a Player in the Game of Blackjack

The card game of blackjack is played with at least two players, one of whom is also a dealer. The object of the game is to receive cards from the deck and play to a count of 21 without going over 21. A card's point equals its rank, but all face cards are 10 points, and an ace can count as either 1 or 11 points as needed. At the beginning of the game, the dealer and the player each receive two cards from the deck. The player can see both of her cards and just one of the dealer's cards initially. The player then "hits" or takes one card at a time until her total exceeds 21 (a "bust" loss), or she "passes" (stops taking cards). When the player passes, the dealer reveals his other card and must keep taking cards until his total is greater than or equal to 17. If the dealer's final total is greater than 21, he also loses. Otherwise, the player with the higher point total wins, or else there is a tie.

A computer program that plays this game can use a **Dealer** object and a **Player** object. The dealer's moves are completely automatic, whereas the player's moves (decisions to pass or hit) are partly controlled by a human user. A third

object belonging to the **Blackjack** class sets up the game and manages the interactions with the user. The **Deck** and **Card** classes developed earlier are also included. A class diagram of the system is shown in Figure 8.4.



[FIGURE 8.4] The classes in the blackjack game application

Here is a sample run of the program:

```

>>> from blackjack import Blackjack
>>> game = Blackjack()
>>> game.play()
Player:
2 of Spades, 5 of Spades
7 points
Dealer:
5 of Hearts
Do you want a hit? [y/n]: y
Player:
2 of Spades, 5 of Spades, King of Hearts
17 points
Do you want a hit? [y/n]: n
Dealer:
5 of Hearts, Queen of Hearts, 7 of Diamonds
22 points
Dealer busts and you win
  
```


When a **Player** object is created, it receives two cards. A **Player** object can be hit with another card, asked for the points in its hand, and asked for its string representation. Here is the code for the **Player** class, followed by a brief explanation:

```
from cards import Deck, Card

class Player(object):
    """This class represents a player in
    a blackjack game."""

    def __init__(self, cards):
        self._cards = cards

    def __str__(self):
        """Returns string rep of cards and points."""
        result = ", ".join(map(str, self._cards))
        result += "\n  " + str(self.getPoints()) + " points"
        return result

    def hit(self, card):
        self._cards.append(card)

    def getPoints(self):
        """Returns the number of points in the hand."""
        count = 0
        for card in self._cards:
            if card.rank > 9:
                count += 10
            elif card.rank == 1:
                count += 11
            else:
                count += card.rank
        # Deduct 10 if Ace is available and needed as 1
        for card in self._cards:
            if count <= 21:
                break
            elif card.rank == 1:
                count -= 10
        return count

    def hasBlackjack(self):
        """Dealt 21 or not."""
        return len(self._cards) == 2 and self.getPoints() == 21
```

The problem of computing the points in a player's hand is complicated by the fact that an ace can count as either 1 or 11. The **getPoints** method solves this problem by first totaling the points using an ace as 11. If this initial count is greater than 21, then there is a need to count an ace, if there is one, as a 1. The second loop accomplishes this by counting such aces as long as they are available and needed. The other methods require no comment.

A **Dealer** object also maintains a hand of cards and recognizes the same methods as a **Player** object. However, the dealer's behavior is a bit more specialized. For example, the dealer at first shows just one card, and the dealer repeatedly hits until 17 points are reached or exceeded. Thus, as Figure 8.4 shows, **Dealer** is best defined as a subclass of **Player**. Here is the code for the **Dealer** class, followed by a brief explanation:

```
class Dealer(Player):
    """Like a Player, but with some restrictions."""

    def __init__(self, cards):
        """Initial state: show one card only."""
        Player.__init__(self, cards)
        self._showOneCard = True

    def __str__(self):
        """Return just one card if not hit yet."""
        if self._showOneCard:
            return str(self._cards[0])
        else:
            return Player.__str__(self)

    def hit(self, deck):
        """Add cards while points < 17,
        then allow all to be shown."""
        self._showOneCard = False
        while self.getPoints() < 17:
            self._cards.append(deck.deal())
```

Dealer maintains an extra instance variable, **_showOneCard**, which restricts the number of cards in the string representation to one card at start-up. As soon as the dealer hits, this variable is set to **False**, so all of the cards will be included in the string from then on. The **hit** method actually receives a deck rather than a single card as an argument, so cards may repeatedly be dealt and added to the dealer's list at the close of the game.

The **Blackjack** class coordinates the interactions among the **Deck** object, the **Player** object, the **Dealer** object, and the human user. Here is the code:

```
class Blackjack(object):

    def __init__(self):
        self._deck = Deck()
        self._deck.shuffle()

        # Pass the player and the dealer two cards each
        self._player = Player([self._deck.deal(),
                                self._deck.deal()])
        self._dealer = Dealer([self._deck.deal(),
                                self._deck.deal()])

    def play(self):
        print("Player:\n", self._player)
        print("Dealer:\n", self._dealer)

        # Player hits until user says NO
        while True:
            choice = input("Do you want a hit? [y/n]: ")
            if choice in ("Y", "y"):
                self._player.hit(self._deck.deal())
                points = self._player.getPoints()
                print("Player:\n", self._player)
                if points >= 21:
                    break
            else:
                break
        playerPoints = self._player.getPoints()
        if playerPoints > 21:
            print("You bust and lose")
        else:
            # Dealer's turn to hit
            self._dealer.hit(self._deck)
            print("Dealer:\n", self._dealer)
            dealerPoints = self._dealer.getPoints()

            # Determine the outcome
            if dealerPoints > 21:
                print("Dealer busts and you win")
            elif dealerPoints > playerPoints:
                print("Dealer wins")
            elif dealerPoints < playerPoints and playerPoints <= 21:
                print("You win")
            elif dealerPoints == playerPoints:
                if self._player.hasBlackjack() and\
```

continued

```

        not self._dealer.hasBlackjack():
            print("You win")
    elif not self._player.hasBlackjack() and \
        self._dealer.hasBlackjack():
            print("Dealer wins")
    else:
        print("There is a tie")

```

8.5.4 Polymorphic Methods

As we have seen in our two examples, a subclass inherits data and methods from its parent class. We would not bother subclassing unless the two classes shared a substantial amount of **abstract behavior**. By this term, we mean that the classes have similar sets of methods or operations. A subclass usually adds something extra, such as a new method or a data attribute, to the ensemble provided by its superclass. A new data attribute is included in both of our examples, and a new method is included in the first one.

In some cases, the two classes have the same interface, or set of methods available to external users. In these cases, one or more methods in a subclass override the definitions of the same methods in the superclass to provide specialized versions of the abstract behavior. Like any object-oriented language, Python supports this capability with **polymorphic methods**. The term “polymorphic” means many bodies, and applies to two methods that have the same header, but have different definitions in different classes. Two examples are the **withdraw** method in the bank account hierarchy and the **hit** method in the blackjack player hierarchy. The `__str__` method is a good example of a polymorphic method that appears throughout Python’s system of classes.

Like other abstraction mechanisms, polymorphic methods make code easier to understand and use, because the programmer does not have to remember so many different names.

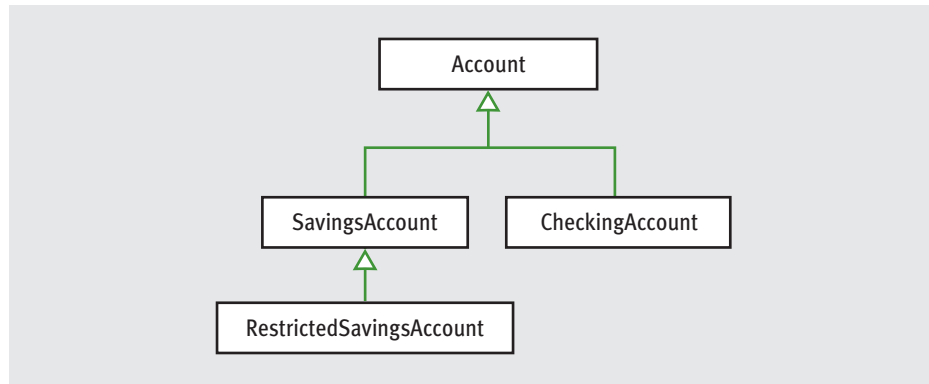
8.5.5 Abstract Classes

An **abstract class** includes data and methods common to its subclasses, but is never instantiated. For example, checking accounts and savings accounts have similar attributes and behavior. The data and methods that they have in common can be placed in an abstract class named **Account**. The **SavingsAccount** and **CheckingAccount** classes can then extend the **Account** class and access these common resources by inheritance (see the UML diagram in Figure 8.5). Any

special behavior or attributes can then be added to these two subclasses.

SavingsAccount and **CheckingAccount** are also known as **concrete classes**.

Unlike concrete classes, an abstract class such as **Account** is never instantiated.



[FIGURE 8.5] An abstract class and three concrete classes

8.5.6

The Costs and Benefits of Object-Oriented Programming

Whenever you learn a new style of programming, you sooner or later become acquainted with its costs and benefits. To hasten this process, we conclude this section by comparing several programming styles, all of which have been used in this book.

The approach with which this book began is called **imperative programming**. Code in this style consists of input and output statements, assignment statements, and control statements for selection and iteration. The name derives from the idea that a program consists of a set of commands to the computer, which responds by performing such actions as manipulating data values in memory. This style is appropriate for writing very short code sequences that accomplish simple tasks, such as solving the problems that were introduced in Chapters 1 through 5 of this book.

However, as problems become more complex, the imperative programming style does not scale well. In particular, the number of interactions among statements that manipulate the same data variables quickly grows beyond the point of comprehension of a human programmer who is trying to verify or maintain the code.

As we saw in Chapter 6, you can mitigate some of this complexity by embedding sequences of imperative code in function definitions or subprograms. It then becomes possible to decompose complex problems into simpler subproblems that can be solved by these subprograms. In other words, the use of subprograms reduces the number of program components that one must keep track of. Moreover, when each subprogram has its own temporary variables and receives data from the surrounding program by means of explicit parameters, the number of possible dependencies and interactions among program components also decreases. The use of cooperating subprograms to solve problems is called **procedural programming**.

Although procedural programming takes a step in the direction of controlling program complexity, it simply masks and ultimately recapitulates the problems of imperative programming at a higher level of abstraction. When many subprograms share and modify a common data pool, as they did in some of our early examples, it becomes difficult once again for the programmer to keep track of all of the interactions among the subprograms during verification and maintenance.

One cause of this problem is the use of the assignment statement to modify data. Some computer scientists have developed a style of programming that dispenses with assignment altogether. This radically different approach, called **functional programming**, views a program as a set of cooperating functions. A function in this sense is a highly restricted subprogram. Its sole purpose is to transform the data in its arguments into other data, its returned value. Because assignment does not exist, functions perform computations by either evaluating expressions or calling other functions. Selection is handled by a conditional expression, which is like an **if-else** statement that returns a value, and iteration is implemented by recursion. By restricting how functions can use data, this very simple model of computation dramatically reduces the conceptual complexity of programs. However, some argue that this style of programming does not conveniently model situations where data objects must change their state.

Object-oriented programming attempts to control the complexity of a program while still modeling data that change their state. This style divides up the data into relatively small units called objects. Each object is then responsible for managing its own data. If an object needs help with its own tasks, it can call upon another object or rely on methods defined in its superclass. The main goal is to divide responsibilities among small, relatively independent or loosely coupled components. Cooperating objects, when they are well designed, decrease the likelihood that a system will break when changes are made within a component.

Although object-oriented programming has become quite popular, it can be overused and abused. Many small and medium-sized problems can still be solved effectively, simply, and, most important, quickly using any of the other three styles of programming mentioned here, either individually or in combination.

The solutions of problems, such as numerical computations, often seem contrived when they are cast in terms of objects and classes. For other problems, the use of objects is easy to grasp, but their implementation in the form of classes reflects a complex model of computation with daunting syntax and semantics. Finally, hidden and unpleasant interactions can lurk in poorly designed inheritance hierarchies that resemble those afflicting the most brittle procedural programs.

To conclude, whatever programming style or combination of styles you choose to solve a problem, good design and common sense are essential.

8.5 Exercises

- 1 What are the benefits of having class **B** extend or inherit from class **A**?
- 2 Describe what the `__init__` method should do in a class that extends another class.
- 3 Class **B** extends class **A**. Class **A** defines an `__str__` method that returns the string representation of its instance variables. Class **B** defines a single instance variable named `_age`, which is an integer. Write the code to define the `__str__` method for class **B**. This method should return the combined string information from both classes. Label the data for `_age` with the string "Age: ".

Summary

- A simple class definition consists of a header and a set of method definitions. Several related classes can be defined in the same module. Each element, a module, a class, and a method, can have a separate docstring associated with it.
- In addition to methods, a class can also include instance variables. These represent the data attributes of the class. Each instance or object of a class has its own chunk of memory storage for the values of its instance variables.
- The constructor or `__init__` method is called when a class is instantiated. This method initializes the instance variables. The method can expect required and/or optional arguments to allow the users of the class to provide initial values for the instance variables.

- A method contains a header and a body. The first parameter of a method is always the reserved word **self**. This parameter is bound to the object with which the method is called, so that the code within the method can reference that particular object.
- An instance variable is introduced and referenced like any other variable, but is always prefixed with **self**. The scope of an instance variable is the body of the enclosing class definition, whereas its lifetime is the lifetime of the object associated with it.
- Some standard operators can be overloaded for use with new classes of objects. One overloads an operator by defining a method that has the corresponding name.
- When a program can no longer reference an object, it is considered dead, and its storage is recycled by the garbage collector.
- A class variable is a name for a value that all instances of a class share in common. It is created and initialized when a class is defined and must be accessed by using the class name, a dot, and the variable name.
- Pickling is the process of converting an object to a form that can be saved to permanent file storage. Unpickling is the inverse process.
- The **try-except** statement is used to catch and handle exceptions that might be raised in a set of statements.
- The three most important features of object-oriented programming are encapsulation, inheritance, and polymorphism. All three features simplify programs and make them more maintainable.
- Encapsulation restricts access to an object's data to users of the methods of its class. This helps to prevent indiscriminant changes to an object's data.
- Inheritance allows one class to pick up the attributes and behavior of another class for free. The subclass may also extend its parent class by adding data and/or methods or modifying the same methods. Inheritance is a major means of reusing code.
- Polymorphism allows methods in several different classes to have the same headers. This reduces the need to learn new names for standard operations.
- A data model is a set of classes that are responsible for managing the data of a program. A view is a set of classes that are responsible for presenting information to a human user and handling user inputs. The model/view pattern structures software systems using these two sets of components.

REVIEW QUESTIONS

- 1 An instance variable refers to a data value that
 - a is owned by a particular instance of a class and no other
 - b is shared in common and can be accessed by all instances of a given class
- 2 The name used to refer to the current instance of a class within the class definition is
 - a **this**
 - b **other**
 - c **self**
- 3 The purpose of the `__init__` method in a class definition is to
 - a build and return a string representation of the instance variables
 - b set the instance variables to initial values
- 4 A method definition
 - a can have zero or more parameter names
 - b always must have at least one parameter name, called **self**
- 5 The scope of an instance variable is
 - a the statements in the body of the method where it is introduced
 - b the entire class in which it is introduced
 - c the entire module where it is introduced
- 6 An object's lifetime ends
 - a several hours after it is created
 - b when it can no longer be referenced anywhere in a program
 - c when its data storage is recycled by the garbage collector
- 7 A class variable is used for data that
 - a all instances of a class have in common
 - b each instance owns separately
- 8 Class **B** is a subclass of class **A**. The `__init__` methods in both classes expect no arguments. The call of class **A**'s `__init__` method in class **B** is
 - a **A.__init__()**
 - b **A.__init__(self)**

- 9 The easiest way to save objects to permanent storage is to
 - a convert them to strings and save this text to a text file
 - b pickle them using the **pickle** function **save**
- 10 A polymorphic method
 - a has a single header but different bodies in different classes
 - b creates harmony in a software system

PROJECTS

- 1 Add methods to the **Student** class that compare two **Student** objects. One method should test for equality. The other methods should support the other possible comparisons. In each case, the method returns the result of the comparison of the two students' names.
- 2 This project assumes that you have completed Project 1. Place several **Student** objects into a list and shuffle it. Then run the **sort** method with this list and display all of the students' information.
- 3 The **str** method of the **Bank** class returns a string containing the accounts in random order. Design and implement a change that causes the accounts to be placed in the string by order of name. (*Hint:* You will also have to define some methods in the **SavingsAccount** class.)
- 4 The ATM program allows a user an indefinite number of attempts to log in. Fix the program so that it displays a message that the police will be called after a user has had three successive failures. The program should also shut down the bank when this happens.
- 5 Develop a terminal-based program that allows a bank manager to manipulate the accounts in a bank. This menu-driven program should include all of the relevant options, such as adding a new account, removing an account, and editing an account.
- 6 A simple software system for a library models a library as a collection of books and patrons. A patron can have at most three books out on loan at any given time. Each book has a title, an author, a patron to whom it has been checked out, and a list of patrons waiting for that book to be returned. When a patron wants to borrow a book, that patron is

automatically added to the book's wait list if the book is already checked out. When a patron returns a book, it is automatically loaned to the first patron on its wait list who can check out a book. Each patron has a name and the number of books that patron has currently checked out. Develop the classes **Book** and **Patron** to model these objects. Think first of the interface or set of methods to be used with each class, and then choose appropriate data structures for the state of the objects. Also write a short script to test these classes.

- 7 Develop a **Library** class that can manage the books and patrons from Project 6. This class should include methods for adding, removing, and finding books and patrons.
- 8 Develop a **Manager** class that provides a menu-driven command processor for managing a library of the type developed in Project 7.
- 9 The **Doctor** program described in Chapter 5 combines the data model of a doctor and the operations for handling user interaction. Restructure this program according to the model/view pattern so that these areas of responsibility are assigned to separate sets of classes. The program should include a **Doctor** class with an interface that allows one to obtain a greeting, a signoff message, and a reply to a patient's string. The rest of the program handles the user's interactions with the **Doctor** object.
- 10 Geometric shapes can be modeled as classes. Develop classes for line segments, circles, and rectangles. Each shape object should contain a **Turtle** object and a color that allow the shape to be drawn in a Turtle graphics window (see Chapter 7 for details). Factor the code for these features (instance variables and methods) into an abstract **Shape** class. The **Circle**, **Rectangle**, and **Line** classes are all subclasses of **Shape**. These subclasses include the other information about the specific types of shapes, such as a radius or a corner point and a **draw** method. Write a script that uses several instances of the different shape classes to draw a house and a stick figure.

[CHAPTER] 9

Graphical User Interfaces

After completing this chapter, you will be able to:

- Structure a GUI-based program using the model/view/controller pattern
- Instantiate and lay out different types of window objects, such as labels, entry fields, and command buttons, in a window's frame
- Define methods that handle events associated with window objects
- Organize sets of window objects in nested frames

Most people do not judge a book by its cover. They are interested in its contents, not its appearance. However, users judge a software product by its user interface because they have no other way to access its functionality. With the exception of Chapter 7, in which we explored graphics and image processing, this book has focused on programs that present a terminal-based user interface. Although this type of user interface is perfectly adequate for some applications, most modern computer software employs a **graphical user interface** or **GUI**. A GUI displays text as well as small images (called icons) that represent objects such as directories, files of different types, command buttons, and drop-down menus. In addition to entering text at the keyboard, the user of a GUI can select an icon with a pointing device, such as a mouse, and move that icon around on the display. Commands can be activated by pressing the enter key or control keys, by pressing a command button, or by selecting a drop-down menu item with the mouse. Put more simply, a GUI displays all information, including text, graphically to its users, and allows them to manipulate this information directly with a pointing device.



In this chapter, you learn how to develop GUIs. The transition to GUIs involves making two adjustments to your thinking. First, the structure of a GUI program differs significantly from that of a terminal-based program. Second, a GUI program is event driven, meaning that it is inactive until the user clicks a button or selects a menu option. In contrast, a terminal-based program maintains constant control over the interactions with the user. Put differently, a terminal-based program prompts the user to enter successive inputs, whereas a GUI program allows the user to enter inputs in any order and waits for the user to press a command button or select a menu option. This distinction will become clearer as you read this chapter.

9.1

The Behavior of Terminal-Based Programs and GUI-Based Programs

We begin by examining the look and behavior of two different versions of the same program from a user's point of view. This program, first introduced as Programming Project 4 in Chapter 3, computes and displays the total distance traveled by a ball, given three inputs—the initial height from which it is dropped, its bounciness index, and the number of bounces. The total distance traveled for a single bounce is the sum of the distance down and the distance back up. The user may enter the inputs any number of times before quitting the program. The first version of the **bouncy** program includes a terminal-based user interface, whereas the second version uses a graphical user interface. Although both programs perform exactly the same function, their behavior, or look and feel, from a user's perspective are quite different.

9.1.1

The Terminal-Based Version

The terminal-based version of the **bouncy** program displays a greeting and then a menu of command options. The user is prompted for a command number and then enters a number from the keyboard. The program responds by either terminating execution, prompting for the information for a bouncing ball, or printing a message indicating an unrecognized command. After the program processes a command, it displays the menu again, and the same process starts over. A sample session with this program is shown in Figure 9.1.

```
Welcome to the bouncy program!
```

```
1 Compute the total distance
2 Quit the program
```

```
Enter a number: 1
```

```
Enter the initial height: 10
Enter the bounciness index: .6
Enter the number of bounces: 2
```

```
The total distance is 25.6
```

```
1 Compute a distance
2 Quit the program
```

```
Enter a number: 2
```

[FIGURE 9.1] A session with the terminal-based **bouncy** program

This terminal-based user interface has several obvious effects on its users:

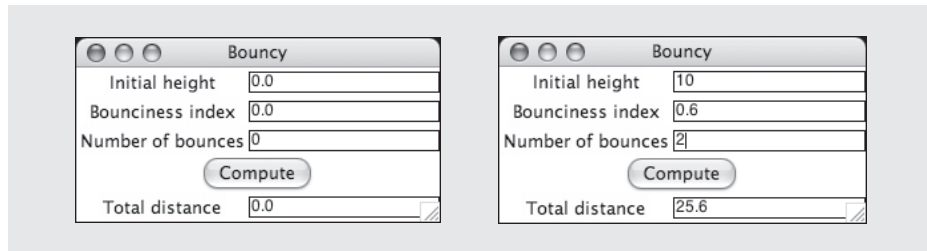
- The user is constrained to reply to a definite sequence of prompts for inputs. Once an input is entered, there is no way to back up and change it.
- To obtain results for a different set of input data, the user must wait for the command menu to be displayed again. At that point, the same command and all of the other inputs must be re-entered.
- The user can enter an unrecognized command.

Each of these effects poses a problem for users that can be solved by converting the interface to a GUI.

9.1.2 The GUI-Based Version

The GUI-based version of the **bouncy** program displays a window that contains various components, also called **window objects** or **widgets**. Some of these components look like text, while others provide visual cues as to their use. Figure 9.2 shows snapshots of a sample session with this version of the program. The snapshot on the left shows the interface at program start-up, whereas the snapshot on

the right shows the interface after the user has entered inputs and selected the **Compute** button. This program was run on a Macintosh; on a Windows- or Linux-based PC, the windows look slightly different.



[FIGURE 9.2] A GUI-based bouncy program

The Bouncy window in Figure 9.2 contains the following components:

- A **title bar** at the top of the window. This bar contains the title of the program, “Bouncy.” It also contains three colored circles. Each circle is a **command button**. The user can use the mouse to click the left circle to quit the program, the middle circle to minimize the window, or the right circle to zoom the window. The user can also drag the window around the screen by holding the left mouse button on the title bar and dragging the mouse.
- A set of **labels** along the left side of the window. These are text elements that describe the inputs and outputs. For example, “Initial height” is one label.
- A set of **entry fields** along the right side of the window. These are boxes within which the program can output text or receive it as input from the user. The first three entry fields will be used for inputs, while the last field will be used for the output. At program start-up, the fields contain default values, as shown in the window on the left side of Figure 9.2.
- A single command button labeled **Compute**. When the user uses the mouse to press this button, the program responds by using the data in the three input fields to compute the total distance. This result is then displayed in the output field. Sample input data and the corresponding output are shown in the window on the right side of Figure 9.2.
- The user can also alter the size of the window by holding the mouse on its lower-right corner and dragging in any direction.

Although this review of features might seem tedious to anyone who regularly uses GUI-based programs, a careful inventory is necessary for the programmer

who builds them. Also, a close study of these features reveals the following effects on users:

- The user is not constrained to enter inputs in a particular order. Before she presses the **Compute** button, she can edit any of the data in the three input fields.
- Running different data sets does not require re-entering all of the data. The user can edit just one or two values and press the **Compute** button.
- The user cannot enter an unrecognized command. Each command option is presented as a virtual button to be pressed.

When we compare the effects of the two interfaces on users, the GUI seems to be a definite improvement on the terminal-based user interface. The improvement is even more noticeable as the number of command options increases and the information to be presented grows in quantity and complexity.

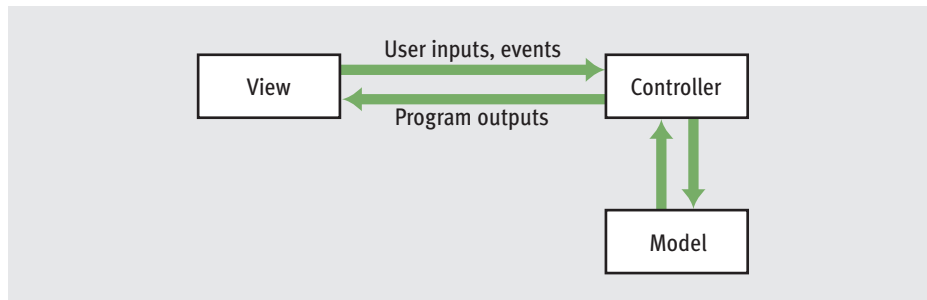
9.1.3 Event-Driven Programming

Rather than guide the user through a series of prompts, a GUI-based program opens a window and waits for the user to manipulate window objects with the mouse. These user-generated **events**, such as mouse clicks, trigger operations in the program to respond by pulling in inputs, processing them, and displaying results. This type of software system is **event-driven**, and the type of programming used to create it is called **event-driven programming**.

Like any complex program, an event-driven program is developed in several steps. In the analysis step, the types of window objects and their arrangement in the window are determined. Because GUI-based programs are almost always object-oriented, this becomes a matter of choosing among GUI component classes available in the programming language or inventing new ones if needed. Graphic designers and cognitive psychologists might be called in to assist in this phase, if the analysts do not already possess this type of expertise.

GUI-based programs also adhere to the model/view pattern that we introduced in Chapter 8. This pattern separates the resources and responsibilities for managing the data model from those concerned with displaying it and interacting with the users. To a certain extent, the number, types, and arrangement of the window objects depend on the nature of the information to be displayed and also depend on the set of commands that will be available to the user for manipulating that information. Thus, the developers of the GUI also have to converse with the developers of the program's data model.

In the design of a GUI-based program, a third set of resources called the **controller** often handles the interactions between a program's data model and its view. The relationships between these three sets of resources, also called the **model/view/controller pattern** or **MVC**, are depicted in Figure 9.3.



[FIGURE 9.3] The model/view/controller pattern

Let us return to the example of the **bouncy** program to see how the MVC pattern works. The GUI in this program consists of the window and its components, including the labeled entry fields and the **Compute** button. The data model, which admittedly is not very complex, consists of a function that receives three numeric arguments and returns the total distance. When the user presses the **Compute** button, a hidden controller object automatically detects this event and triggers or calls a controller method. This method in turn fetches the input values from the input fields and passes them to the data model for processing. When the data model returns its result, the controller method sends it to the output field to be displayed. Ideally, the view knows nothing about the data model, and the data model knows nothing about the view. The controller conducts the conversations between them.

Once the interactions among these resources have been determined, their coding can begin. This phase consists of several steps:

- 1 Define a new class to represent the main application window.
- 2 Instantiate the classes of window objects needed for this application, such as labels, fields, and command buttons.
- 3 Position these components in the window.
- 4 Instantiate the data model and provide for the display of any default data in the window objects.

- 5 Register controller methods with each window object in which an event relevant to the application might occur.
- 6 Define these controller methods.
- 7 Define a **main** function that instantiates the window class and runs the appropriate method to launch the GUI.

In coding the program, you could initially skip steps 4–6, which concern the controller and the data model, to develop and refine the view. This would allow you to preview the window and its layout, even though the command buttons and other GUI elements lack functionality.

In the sections that follow, we explore these elements of GUI-based, event-driven programming with examples in Python.

9.1 Exercises

- 1 Describe two fundamental differences between terminal-based user interfaces and GUIs.
- 2 Describe the responsibilities of the model, view, and controller in the MVC pattern.
- 3 Give an example of one application for which a terminal-based user interface is adequate and one example that lends itself best to a GUI.

9.2 Coding Simple GUI-Based Programs

In this section, we show some examples of simple GUI-based programs in Python. There are many libraries and toolkits of GUI components available to the Python programmer, but in this chapter we use just two: **tkinter** and **tkinter.messagebox**. Both are standard modules that come with any Python installation. **tkinter** includes classes for windows and numerous types of window objects. **tkinter.messagebox** includes functions for several standard pop-up dialog boxes. We start with some short demo programs that illustrate each type of GUI component, and, in later sections, we develop some examples with more significant functionality.

9.2.1 Windows and Labels

Our first demo program defines a class for a main window that displays a greeting. In all of our GUI-based programs, this class extends **tkinter**'s **Frame** class. The **Frame** class provides the basic functionality for any window, such as the command buttons in the title bar. Here is the code, followed by Figure 9.4, which shows a screenshot of the window:

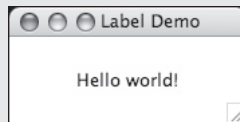
```
from tkinter import *

class LabelDemo(Frame):

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Label Demo")
        self.grid()
        self._label = Label(self, text = "Hello world!")
        self._label.grid()

def main():
    """Instantiate and pop up the window."""
    LabelDemo().mainloop()

main()
```



[FIGURE 9.4] Displaying a text label in a window

The **LabelDemo** class's **__init__** method includes five statements that perform the following tasks:

- 1 Run **Frame**'s **__init__** method to automatically initialize any variables defined in the **Frame** class.
- 2 Reset the window's title. In this line of code, **self.master** is an instance variable defined in the **Frame** class. This variable refers to the root window.

This window in turn has an instance variable named **title**, which by default is an empty string.

- 3 Use the **grid** method to set the window's layout manager to a **grid layout**. A grid layout allows the programmer to place components in the cells of an invisible grid in the window. The nature and purpose of this grid will become clear in upcoming examples that contain multiple window objects.
- 4 Create the only window component, a **Label** object. When a component is created, its constructor expects the **parent component** as an argument. In this case, the parent of the label is the **LabelDemo** instance, or **self**. The other arguments can be keyword arguments that specify the component's attributes. In this example, the label receives a **text** attribute, whose value is a string of text to be displayed when the label is displayed in the window.
- 5 Use the **grid** method again to position the label in the window's grid. In this case, the label will appear centered in the window.

The GUI is launched in the **main** method. This method instantiates **LabelDemo** and calls its **mainloop** method. This method pops up the window and waits for user events. At this point, the **main** method in our own code quits, because the GUI is running a hidden, event-driven loop in a separate process. This part of the program does not vary much from application to application, so we omit it from the next few examples.

If you are running the program as a script from a terminal prompt, pressing the window's close button will quit the program normally. If you are launching the program from an IDLE window, you should close the shell window as well as the program's window to terminate the process that is running the GUI.

9.2.2 Displaying Images

Our next example modifies the first one slightly, so that the program displays an image and a caption. We use labels for both components. To create a label with an image, two steps are required. First, the **__init__** method creates an instance of the class **PhotoImage** from a GIF file on disk (remember that the image file must be in the current working directory). Then the new label's **image** attribute is set to the **PhotoImage** object. The label for the caption is set up with a **text** attribute, as described earlier. The image label is placed in the grid before the text label. The resulting labels are centered in a column in the window. Here is the code for

a program that displays a captioned image of Smokey the cat, followed by a screenshot of the window in Figure 9.5:

```
from tkinter import *

class ImageDemo(Frame):

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Image Demo")
        self.grid()
        self._image = PhotoImage(file = "smokey.gif")
        self._imageLabel = Label(self, image = self._image)
        self._imageLabel.grid()
        self._textLabel = Label(self, text = "Smokey the cat")
        self._textLabel.grid()
```



[FIGURE 9.5] Displaying a captioned image

9.2.3 Command Buttons and Responding to Events

Command buttons are created and placed in a window in the same manner as labels. Also like labels, a button can display either text or an image. When the **Button** object receives a **text** attribute, it displays the associated string. When the button receives an **image** attribute, it provides a clickable image.

To activate a button and enable it to respond to mouse clicks, you must set its **command** attribute to an event-handling method. This is done in the main window's **__init__** method when the button is created. The value of the command attribute is just the variable that refers to the event-handling method. The method itself is then defined later in the main window class.

Here is the code for an example program that allows the user to press a button to change a label's text. The text alternates between **"Hello"** and **"Goodbye"**. Figure 9.6 shows the two states of the window.

```
from tkinter import *

class ButtonDemo(Frame):

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Button Demo")
        self.grid()
        self._label = Label(self, text = "Hello")
        self._label.grid()
        self._button = Button(self,
                               text = "Click me",
                               command = self._switch)
        self._button.grid()

    def _switch(self):
        """Event handler for the button."""
        if self._label["text"] == "Hello":
            self._label["text"] = "Goodbye"
        else:
            self._label["text"] = "Hello"
```



[FIGURE 9.6] When the user presses the Click me button, the message changes from “Hello” to “Goodbye”

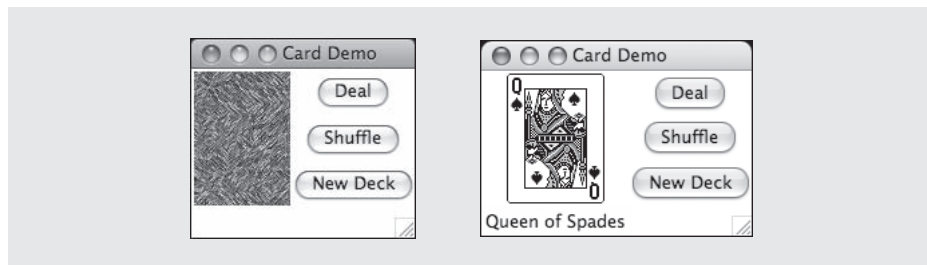
Note that the **_switch** method examines the **text** attribute of the label and sets it to the appropriate value. The attributes of each window component are actually stored in a dictionary, so the notation for examining them and modifying them includes the subscript operator with the name of the attribute as the key.

In programs that use several buttons, each button has its own event-handling method. The standard procedure in the **__init__** method is to create the buttons, set their **command** attribute, and lay them out in the grid. Later in the window class, the event-handling methods for all of the buttons are then defined.

These methods together make up the controller part of the MVC pattern discussed in Section 9.1.

9.2.4 Viewing the Images of Playing Cards

Modern game-playing programs provide graphical displays of the characters and the setting of a game. Games that use playing cards display images of the cards. We now present a program that allows the user to view the cards in a deck. The GUI is shown in Figure 9.7. At start-up, the window displays the back of a card, along with three command buttons. The user can select a command to deal a card, shuffle the deck, or obtain a new deck. As each new card is dealt, an image of its face and the text of its rank and suit are displayed. The user can continue to deal cards until the deck becomes empty.



[FIGURE 9.7] A GUI for viewing playing cards

Images of playing cards are available as open source on many Web sites. On such sites, the filenames for the images typically indicate the rank and suit of the card. In this example, the filename for the image of the queen of spades is **12s.gif**. If the entire set of files is located in a folder named **DECK**, the path to this filename is actually **DECK/12s.gif**.

The GUI for this program will have to obtain the filename of the image for each card displayed, as well as the filename of the image for the backside of each card. A couple of changes to the **Card** class defined in Section 8.3.9 will provide this information. We add an instance variable for the filename of the card's image on disk. At instantiation, the **__init__** method uses the rank and suit information to build a filename and sets a new instance variable to this string. Thus, each card's image filename can be accessed by using its **fileName** attribute. There is also a single image that represents the backside of all of the cards in a file named

b.gif. A new class variable, **BACK_NAME**, is defined to be this filename. Here is the code for these revisions to the **Card** class:

```
BACK_NAME = 'DECK/b.gif'

def __init__(self, rank, suit):
    """Creates a card with the given rank, suit, and
    image filename."""
    self.rank = rank
    self.suit = suit
    self.fileName = 'DECK/' + str(rank) + suit[0].lower() + '.gif'
```

The main window class is called **CardDemo**. It maintains instance variables for the deck of cards, the image of each card's backside, and the image of the current card. The back image is loaded at start-up and does not change. The card image is initially **None** before the user deals a card. **Label** components are then set up for the image and the text of a card. The image label initially holds the backside image, whereas the text label holds the empty string. Three command buttons are created and added to the window.

The window components are now laid out in explicit rows and columns in the window's grid. There are two columns and four rows. The left column contains the card image and its caption, whereas the right column contains the three command buttons. The rows and columns of the grid are numbered from 0. Thus, the card image in the upper-left corner is located at position (0, 0), and the topmost command button occupies position (0, 1). The **grid** method specifies these positions by receiving values for the **row** and **column** attributes. Take care to position each component properly. Drawing a sketch of the grid with example coordinates can help with the design of a layout.

Although the card image lies in the first row of the grid, it must occupy three rows to align with the three buttons in the column to its right. You can stretch a window component across several rows by specifying the value of the **rowspan** attribute. Thus, the card image receives a **rowspan** of 3.

There are three event-handling methods:

- 1 The method **_shuffle** simply shuffles the deck.
- 2 The method **_deal** requests the next card from the deck. If this card is not **None**, its image is loaded and displayed, and its string representation is also obtained and displayed.
- 3 The method **_new** restores all of the data and the GUI to their initial states.

If the card just dealt equals **None**, the deck is empty, so method **_new** is called to return the user to the initial state of the demo.

Here is the code:

```
from tkinter import *
from cards import Card, Deck

class CardDemo(Frame):

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Card Demo")
        self.grid()
        self._deck = Deck()
        self._backImage = PhotoImage(file = Card.BACK_NAME)
        self._cardImage = None
        self._imageLabel = Label(self, image = self._backImage)
        self._imageLabel.grid(row = 0, column = 0, rowspan = 3)
        self._textLabel = Label(self, text = "")
        self._textLabel.grid(row = 3, column = 0)

        self._dealButton = Button(self,
                                   text = "Deal",
                                   command = self._deal)
        self._dealButton.grid(row = 0, column = 1)
        self._shuffleButton = Button(self,
                                      text = "Shuffle",
                                      command = self._shuffle)
        self._shuffleButton.grid(row = 1, column = 1)
        self._newButton = Button(self,
                                  text = "New Deck",
                                  command = self._new)
        self._newButton.grid(row = 2, column = 1)

    def _deal(self):
        """If the deck is not empty, deals and displays the
        next card. Otherwise, returns the program to its
        initial state."""
        card = self._deck.deal()
        if card != None:
            self._cardImage = PhotoImage(file = card.fileName)
            self._imageLabel["image"] = self._cardImage
            self._textLabel["text"] = str(card)
        else:
            self._new()

    def _shuffle(self):
        self._deck.shuffle()
```

continued

```

def _new(self):
    """Returns the program to its initial state."""
    self._deck = Deck()
    self._cardImage = None
    self._imageLabel["image"] = self._backImage
    self._textLabel["text"] = ""

def main():
    CardDemo().mainloop()

main()

```

9.2.5 Entry Fields for the Input and Output of Text

Anyone who shops on the Web has used a **form filler** to enter a name, password, and credit card number. A form filler consists of labeled **entry fields**, which allow the user to enter and edit a single line of text. A field can also contain text output by a program. **tkinter**'s **Entry** class is used to display an entry field. To facilitate the input and output of floating-point numbers, an **Entry** object is associated with a **container object** of the **DoubleVar** class. This object contains the data value that is displayed in the **Entry** object. The **DoubleVar** object's **set** method is used to output a floating-point number to the associated **Entry** object. Its **get** method is used to input a floating-point number from the associated **Entry** object.

An **Entry** object is set up in two steps. First, its **DoubleVar** object is created. Its default content is 0.0, but its **set** method may be run to give it a different initial value. Then the **Entry** is created with the **DoubleVar** object as the value of its **textvariable** attribute. The contents of the **DoubleVar** object can then be accessed or modified by any event-handler methods. The three types of data container objects that can be used with **Entry** fields are listed in Table 9.1. The methods **get** and **set** are used with all three types of containers.

TYPE OF DATA	TYPE OF DATA CONTAINER
float	DoubleVar
int	IntVar
str (string)	StringVar

[TABLE 9.1] Data container classes for different data types

Our next demo program recasts the **circlearea** program of Programming Project 6 of Chapter 1 as a GUI program. Here is the code, followed by a screenshot of the GUI in Figure 9.8:

```
from tkinter import *
import math

class CircleArea(Frame):

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Circle Area")
        self.grid()

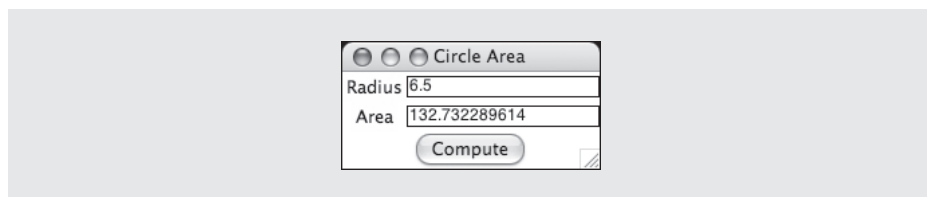
        # Label and field for the radius
        self._radiusLabel = Label(self, text = "Radius")
        self._radiusLabel.grid(row = 0, column = 0)
        self._radiusVar = DoubleVar()
        self._radiusEntry = Entry(self,
                                   textvariable = self._radiusVar)
        self._radiusEntry.grid(row = 0, column = 1)

        # Label and field for the area
        self._areaLabel = Label(self, text = "Area")
        self._areaLabel.grid(row = 1, column = 0)
        self._areaVar = DoubleVar()
        self._areaEntry = Entry(self,
                                   textvariable = self._areaVar)
        self._areaEntry.grid(row = 1, column = 1)

        # The command button
        self._button = Button(self,
                               text = "Compute",
                               command = self._area)
        self._button.grid(row = 2, column = 0, columnspan = 2)

    def _area(self):
        """Event handler for the button."""
        radius = self._radiusVar.get()
        area = radius ** 2 * math.pi
        self._areaVar.set(area)

def main():
    CircleArea().mainloop()
main()
```



[FIGURE 9.8] The `circlearea` program recast as a GUI program

9.2.6 Using Pop-Up Dialog Boxes

GUI-based programs rely extensively on pop-up dialog boxes that display messages, query the user for a Yes/No answer, and so forth. The `tkinter.messagebox` module includes several functions that perform these tasks. Some of these functions are listed in Table 9.2.

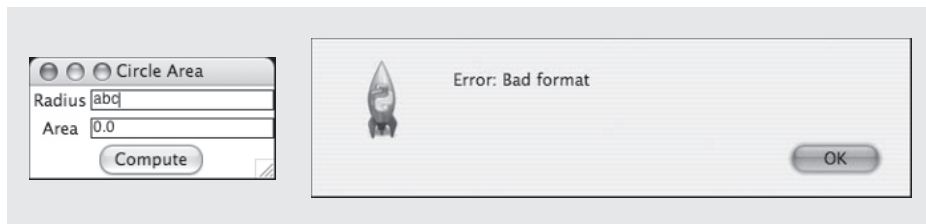
<code>tkinter.messagebox</code> FUNCTION	WHAT IT DOES
<code>askokcancel(title = None, message = None, parent = None)</code>	Asks an OK/Cancel question, returns True if OK is selected, False otherwise.
<code>askyesno(title = None, message = None, parent = None)</code>	Asks a Yes/No question, returns True if Yes is selected, False otherwise.
<code>showerror(title = None, message = None, parent = None)</code>	Shows an error message.
<code>showinfo(title = None, message = None, parent = None)</code>	Shows information.
<code>showwarning(title = None, message = None, parent = None)</code>	Shows a warning message.

[TABLE 9.2] Some `tkinter.messagebox` functions

The keyword arguments for each function can receive values for the dialog box's title, message, and parent component, usually the main window from which the pop-up is launched.

Let's add error handling to the `_area` method in the GUI program for computing the area of a circle. To do this, we can set up the method to use a **try-except** statement that catches a **ValueError**. This type of exception is raised when Python attempts to convert a string with a bad format to a number. If a **ValueError** is raised, the `_area` method pops up an error dialog box to display a message. Here is the code, followed by a screenshot that shows the pop-up in Figure 9.9:

```
def _area(self):
    """Event handler for the button."""
    try:
        radius = self._radiusVar.get()
        area = radius ** 2 * math.pi
        self._areaVar.set(area)
    except ValueError:
        tkinter.messagebox.showerror(message = "Error: Bad format",
                                     parent = self)
```



[FIGURE 9.9] A pop-up dialog box with an error message

9.2 Exercises

- 1 Explain what usually happens in the `__init__` method of a main window class.
- 2 How is the controller set up in a GUI program?
- 3 Describe the procedure for setting up the display of an image in a window.
- 4 Explain how to position a GUI component in a window's grid layout.
- 5 What roles do the **IntVar**, **DoubleVar**, and **StringVar** classes serve in a GUI program?

9.3 Case Study: A GUI-Based ATM

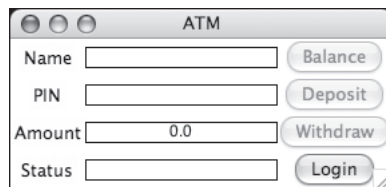
We now pause our survey of GUI components to develop a GUI for a significant application. Case Study 8.4 presented an ATM with a terminal-based user interface. Because we were careful to separate the model from the view in that program, it should now be straightforward to replace that interface with a GUI.

9.3.1 Request

Replace the terminal-based interface of the ATM program with a GUI.

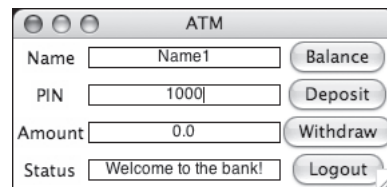
9.3.2 Analysis

The program retains the same functions, but presents the user with a different look and feel. Figure 9.10 shows a sequence of user interactions with the main window.



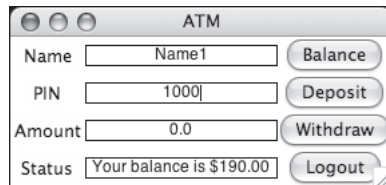
The ATM GUI window titled "ATM" has a light gray background. It contains four text input fields: "Name", "PIN", "Amount" (with "0.0" entered), and "Status". To the right of each field is a button: "Balance" (next to Name), "Deposit" (next to PIN), "Withdraw" (next to Amount), and "Login" (next to Status). The "Status" field is empty.

Interaction 1: Before login



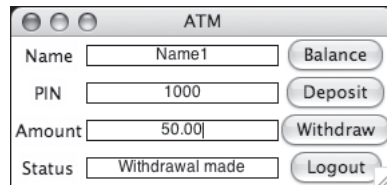
The ATM GUI window titled "ATM" shows the state after login. The "Name" field now contains "Name1", the "PIN" field contains "1000", and the "Amount" field still contains "0.0". The "Status" field now displays "Welcome to the bank!". The buttons are "Balance", "Deposit", "Withdraw", and "Logout" (replacing "Login").

Interaction 2: After login



The ATM GUI window titled "ATM" shows the state after checking the balance. The "Name" field contains "Name1", the "PIN" field contains "1000", and the "Amount" field still contains "0.0". The "Status" field now displays "Your balance is \$190.00". The buttons are "Balance", "Deposit", "Withdraw", and "Logout".

Interaction 3: After balance



The ATM GUI window titled "ATM" shows the state after a withdrawal. The "Name" field contains "Name1", the "PIN" field contains "1000", and the "Amount" field now contains "50.00". The "Status" field now displays "Withdrawal made". The buttons are "Balance", "Deposit", "Withdraw", and "Logout".

Interaction 4: After withdrawal

[FIGURE 9.10] User interactions with the GUI-based ATM

As you can see, the GUI includes three labeled entry fields for the user's name, the user's PIN, and for the amount of money the user wants to withdraw or deposit. A fourth field outputs messages from the program. The **Name** and **PIN** fields are used for inputs during the login process. When the user successfully logs in, the name of the **Login** button changes to **Logout**. The other command buttons perform the named tasks. The field for the amount of money contains the amount to be deposited or withdrawn from the account. The **Status** field displays a greeting, the balance when it is requested, a message signaling the success or failure of a deposit or a withdrawal, and a sign-off message.

There are no new classes, although the **ATM** class now extends the **Frame** class. The model, consisting of the **Bank** and **SavingsAccount** classes, does not change.

9.3.3 Design

Instead of implementing a text-based, menu-driven command processor, the **ATM** class now implements a GUI-based, event-driven command processor.

The `__init__` method receives a **Bank** object from the **main** function and maintains a reference to the current user's account, as before. Its new work lies in creating and laying out the GUI components. There is quite a bit of work here, but the operations are similar to those discussed in earlier examples. The only difference is that three of the four buttons are disabled at program start-up. This is accomplished by setting the button's **state** attribute to the **tkinter** constant **DISABLED**. Otherwise, the `__init__` method requires no further comment.

The helper methods for handling a user's commands now become event-handling methods associated with the command buttons. The two methods that differ from those seen in earlier examples are `_login` and `_logout`.

The `_login` method takes the user's input for the PIN and attempts to find an account for it in the bank. If the account exists, its name is compared to the user's input for the name. If they match, then the following occurs:

- A welcome message is displayed in the status field.
- The login button's **text** attribute is set to **"Logout"**.
- The login button's **command** attribute is set to the `_logout` method, so the user can log out.
- The other command buttons are enabled, using the **tkinter** constant **NORMAL**.

The `_logout` method essentially resets the GUI to its initial state. Here are the details:

- The `_account` variable is reset to `None`.
- The input fields are cleared.
- The login button's `text` attribute is set to `"Login"`.
- The login button's `command` attribute is set to the `_login` method, so another user can log in.
- The other three command buttons are disabled.
- A sign-off message is displayed in the status field.

9.3.4 Implementation (Coding)

Most of the code is in the `__init__` method, where the GUI and its components are set up. The event-handling methods are similar to the methods that handle the basic tasks in the earlier version of the program.

```
"""
File: atm.py

This module defines a GUI-based ATM class and its application.
"""

from bank import Bank, SavingsAccount
from tkinter import *

class ATM(Frame):
    """This class represents GUI-based ATM transactions."""

    def __init__(self, bank):
        """Initialize the frame, widgets, and the data model."""
        Frame.__init__(self)
        self.master.title("ATM")
        self.grid()
        self._bank = bank
        self._account = None

        # Create and add the widgets to the frame.
        # Data containers for entry fields
        self._nameVar = StringVar()
        self._pinVar = StringVar()
        self._amountVar = DoubleVar()
        self._statusVar = StringVar()
```

continued


```

# Labels for entry fields
self._nameLabel = Label(self, text = "Name")
self._nameLabel.grid(row = 0, column = 0)
self._pinLabel = Label(self, text = "PIN")
self._pinLabel.grid(row = 1, column = 0)
self._amountLabel = Label(self, text = "Amount")
self._amountLabel.grid(row = 2, column = 0)
self._statusLabel = Label(self, text = "Status")
self._statusLabel.grid(row = 3, column = 0)

# Entry fields
self._nameEntry = Entry(self,
                        textvariable = self._nameVar,
                        justify = CENTER)
self._nameEntry.grid(row = 0, column = 1)
self._pinEntry = Entry(self,
                      textvariable = self._pinVar,
                      justify = CENTER)
self._pinEntry.grid(row = 1, column = 1)
self._amountEntry = Entry(self,
                          textvariable = self._amountVar,
                          justify = CENTER)
self._amountEntry.grid(row = 2, column = 1)
self._statusEntry = Entry(self,
                          textvariable = self._statusVar,
                          justify = CENTER)
self._statusEntry.grid(row = 3, column = 1)

# Command buttons
self._balanceButton = Button(self, text = "Balance",
                             command = self._getBalance)
self._balanceButton["state"] = DISABLED
self._balanceButton.grid(row = 0, column = 2)
self._depositButton = Button(self, text = "Deposit",
                              command = self._deposit)
self._depositButton["state"] = DISABLED
self._depositButton.grid(row = 1, column = 2)
self._withdrawButton = Button(self, text = "Withdraw",
                               command = self._withdraw)
self._withdrawButton["state"] = DISABLED
self._withdrawButton.grid(row = 2, column = 2)
self._loginButton = Button(self, text = "Login",
                            command = self._login)
self._loginButton.grid(row = 3, column = 2)

# Event-handling methods

```

continued

```

def _getBalance(self):
    self._statusVar.set("Your balance is $%0.2f" % \
        (self._account.getBalance()))

def _deposit(self):
    amount = self._amountVar.get()
    self._account.deposit(amount)
    self._statusVar.set("Deposit made")

def _withdraw(self):
    amount = self._amountVar.get()
    message = self._account.withdraw(amount)
    if message:
        self._statusVar.set(message)
    else:
        self._statusVar.set("Withdrawal made")

def _login(self):
    pin = self._pinVar.get()
    name = self._nameVar.get()
    self._account = self._bank.get(pin)
    if self._account:
        if name == self._account.getName():
            self._statusVar.set("Welcome to the bank!")
            self._loginButton["text"] = "Logout"
            self._loginButton["command"] = self._logout
            self._balanceButton["state"] = NORMAL
            self._depositButton["state"] = NORMAL
            self._withdrawButton["state"] = NORMAL
        else:
            self._statusVar.set("Unrecognized name")
            self._account = None
    else:
        self._statusVar.set("Unrecognized pin")

def _logout(self):
    self._account = None
    self._nameVar.set("")
    self._pinVar.set("")
    self._amountVar.set(0.0)
    self._loginButton["text"] = "Login"
    self._loginButton["command"] = self._login
    self._balanceButton["state"] = DISABLED
    self._depositButton["state"] = DISABLED
    self._withdrawButton["state"] = DISABLED
    self._statusVar.set("Have a nice day!")

```

continued

```

# Top-level functions
def main():
    """Instantiate a bank and use it in an ATM."""
    bank = Bank("bank.dat")
    print("The bank has been loaded")

    atm = ATM(bank)
    print("Running the GUI")
    atm.mainloop()

    bank.save()
    print("The bank has been updated")

def createBank(number = 0):
    """Saves a bank with the specified number of accounts.
    Used during testing."""
    bank = Bank()
    for i in range(number):
        bank.add(SavingsAccount('Name' + str(i),
                                str(1000 + i),
                                100.00))

    bank.save("bank.dat")

```

9.4 Other Useful GUI Resources

Many simple GUI-based applications rely on the resources that we have presented thus far in this chapter. However, as applications become more complex and, in fact, begin to look like the ones we use on a daily basis, other resources must come into play. The layout of GUI components can be specified in more detail, and groups of components can be nested in multiple panes in a window. Paragraphs of text can be displayed in scrolling text boxes. Lists of information can be presented for selection in scrolling list boxes and drop-down menus. The color, size, and style of text and of some GUI components can be adjusted. Finally, GUI-based programs can be configured to respond to various keyboard events and mouse events.

In this section, we provide a brief overview of some of these advanced resources and manipulations, so that you may use them to solve problems in the programming projects.

9.4.1 Colors

The **tkinter** module supports the RGB color system introduced in Chapter 7. In this system, a color consists of three integer components that specify the intensities of red, green, and blue mixed into that color. Each integer ranges from 0 through 255, where 0 means the absence of a color component and 255 means the total saturation of that component. When using **tkinter**, the programmer must express these values using hexadecimal notation. In Python, a hex literal begins with the **#** symbol. The general form of an RGB value is **#rrggbb**. For example, the values **#000000**, **#ffffff**, and **#ff0000** represent the colors black, white, and red, respectively.

tkinter also recognizes some commonly used colors as string values. These include **"white"**, **"black"**, **"red"**, **"green"**, **"blue"**, **"cyan"**, **"yellow"**, and **"magenta"**.

For most GUI components, the programmer can set two color attributes: a foreground color and a background color. The foreground color of a label or an entry field is its text color, whereas the background color is the color of the rectangular area within which the text is displayed. The symbol **fg** names the foreground attribute, and the symbol **bg** names the background attribute. The next code segment sets up a label whose text is red and whose background is light gray:

```
self._exampleLabel = Label(self, text = "Example",
                           fg = "red", bg = "#cccccc")
```

You can also reset a frame's background color. This is done when the **__init__** method of the **Frame** class is called. For example, the following line of code sets the main window's background color to blue:

```
Frame.__init__(self, bg = "blue")
```

Because the background attribute of a label is unrelated to the background attribute of its parent frame, it is a good idea to set both attributes to the same color.

9.4.2 Text Attributes

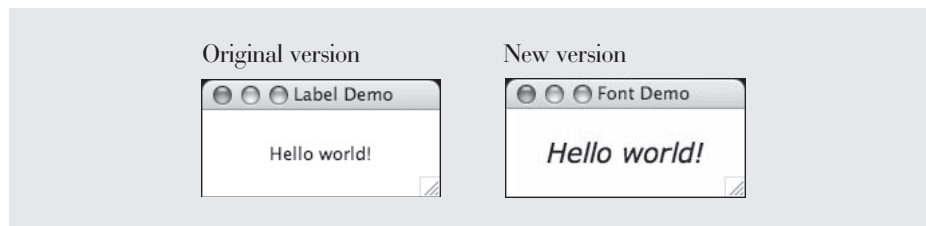
The text displayed in a label, entry field, or button can also have a **type font**. This includes a family, such as Helvetica, a size, such as 24, and a weight, such as bold. Table 9.3 lists the type font attributes and their values.

<code>tkinter.font</code> ATTRIBUTE	VALUES
family	A string, as included in the tuple returned by <code>tkinter.font.families()</code> .
size	An integer specifying the point size.
weight	"bold" or "normal".
slant	"italic" or "roman".
underline	1 or 0.

[TABLE 9.3] Font attributes

The next code segment sets the type font of the label displayed in the first GUI demo program of Section 9.2. The programmer first imports the `tkinter.font` module and instantiates the `Font` class in the `tkinter.font` module. The keyword arguments for the desired attributes are passed to the `Font` constructor. The resulting `Font` object is then used to specify the `font` attribute of the label. Figure 9.11 shows the original window and the new version.

```
font = tkinter.font.Font(family = "Verdana",
                        size = 20, slant = "italic")
self._label = Label(self, font = font, text = "Hello world!")
```



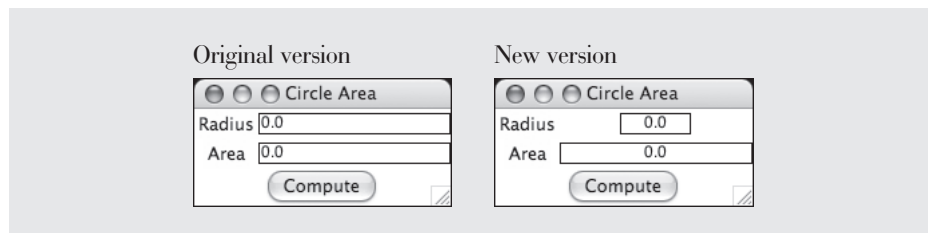
[FIGURE 9.11] Setting a type font

9.4.3 Sizing and Justifying an Entry

It's common to restrict the data in a given entry field to a fixed length, such as a single letter (in the case of a middle initial field) or a nine-digit number (in the case of a Social Security number). For these cases, you can set the width of an

entry field to the appropriate number of columns at instantiation using the **width** attribute. When the width of a field can exceed the length of its content string, you can align this string using the **justify** attribute. The next code segment reduces the width of the radius field to seven columns and centers the text in both fields of the **circlearea** program. The result is shown in Figure 9.12.

```
self._radiusEntry = Entry(self, justify = "center", width = 7,  
                           textvariable = self._radiusVar)  
self._areaEntry = Entry(self, justify = "center",  
                         textvariable = self._areaVar)
```



[FIGURE 9.12] Setting the size and justification of entry fields

9.4.4 Sizing the Main Window

In the GUIs that we have seen thus far, by default the main window shrink-wraps around the components at program start-up. The user can then resize the window by dragging its lower-right corner in any direction. It is also possible for the program to specify the window's initial size and to disable its resizing.

In earlier examples, we set the window's title by using the following expression:

```
self.master.title(<a string>)
```

In this code, **self** refers to the current frame, and **master** refers to the **root window** that contains this frame. Thus, the method **title** is run with the frame's root window to insert the title into the title bar. You can run two other methods, **geometry** and **resizable**, with the root window to affect its sizing.

The method **geometry** expects a string as an argument and uses it to set the size of the main window. This string must be of the form **"widthxheight"**,

where **width** and **height** are integers. Thus, the following expression sets the window's width and height to 200 pixels and 100 pixels, respectively:

```
self.master.geometry("200x100")
```

The window can then be resized at any point under program control or user control.

The method **resizable** expects two integers as arguments. These values enable or disable the window's resizing in the horizontal and vertical directions. A value of 0 disables, whereas a value of 1 enables. Thus, the following expression creates a fixed size window in both directions at start-up:

```
self.master.resizable(0, 0)
```

Neither the program nor the user can resize this window unless this method is run again to enable resizing.

Generally, it is easiest for both the programmer and the user to manage a window that is *not* resizable. Your goal, as a programmer, is to lay out widgets in a manner that is pleasing to the eye and easy to manipulate, so that the user has no reason to resize the window. However, some flexibility might occasionally be warranted. When the window's dimensions must exceed their shrink-wrap defaults, the programmer must master the intricacies of the grid layout. To these we now turn.

9.4.5 Grid Attributes

By default, a newly opened window shrink-wraps around its components and is resizable. When the programmer or the user resizes the window, the components stay shrink-wrapped in their grid, which in turn remains centered within the window. The widgets are also centered within their grid cells.

Occasionally, a widget must be aligned to the left or to the right in its grid cell, the grid must expand with the surrounding window, and/or the components themselves must expand within their grid cells. You can achieve any of these effects by setting the appropriate grid attributes. These attributes are listed in Table 9.4.

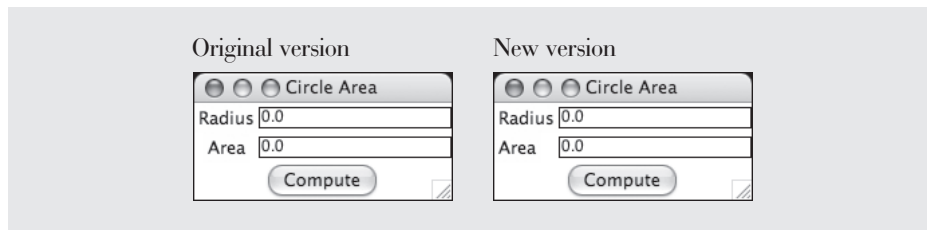
Grid ATTRIBUTE	MEANING
column	The column in which the widget is placed, counting from 0. The default is 0.
columnspan	The number of columns across which the widget is stretched.
ipadx	The number of pixels of horizontal padding added within the boundaries of the widget.
ipady	The number of pixels of vertical padding added within the boundaries of the widget.
padx	The number of pixels of horizontal padding added between the boundaries of the widget and its cell boundaries.
pady	The number of pixels of vertical padding added between the boundaries of the widget and its cell boundaries.
row	The row in which the widget is placed, counting from 0. The default is the next higher-numbered unoccupied row.
rowspan	The number of rows across which the widget is stretched.
sticky	Specifies how to distribute extra space in the widget's cell. Possible values are W , E , N , S , NE , NW , SE , and SW , or combinations thereof, using + . For example, NE aligns the widget in its cell's upper-right corner, whereas W+E allows horizontal expansion.

[TABLE 9.4] Grid attributes

First, we examine how to align widgets within their grid cells. For example, the labels in the **circlearea** program would look better if they were left-aligned. To do this, you specify the **sticky** attributes of both cells as **W** (west), as follows:

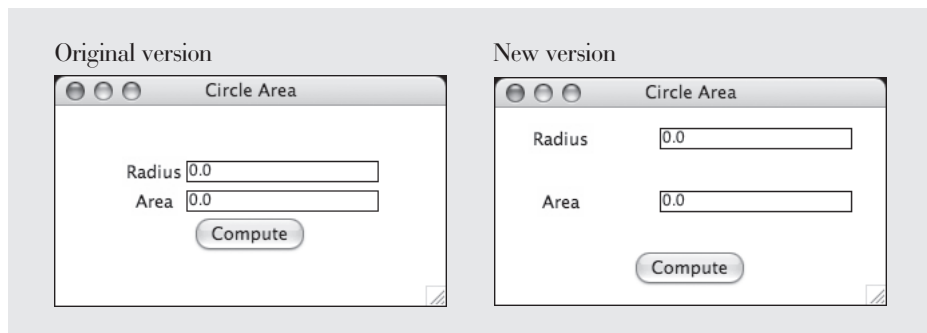
```
self._radiusLabel.grid(row = 0, column = 0, sticky = W)
self._areaLabel.grid(row = 1, column = 0, sticky = W)
```

The result is shown in Figure 9.13.



[FIGURE 9.13] The `circlearea` GUI with left-alignment of labels

Next we consider how to get the grid cells, but not the components within them, to expand with the window. This should have the effect of spreading the widgets apart or drawing them closer together as the window is resized. However, the default behavior of the widgets when the user expands the window is to stay huddled together, in an invisible shrink-wrap, in the center of the window. To override this behavior, you can specify an **expansion weight** on a given row or column of cells. For example, if the weight on row 0 is 1 and the weight on row 1 is 2, then the first row will take one-third of the extra space, and the second row will take two-thirds of the extra space created when the user resizes the window vertically (the total weight of 3 is divided between the two rows as $\frac{1}{3}$ and $\frac{2}{3}$). Likewise, the weights on the columns determine the relative space allotted to them when the window is resized horizontally. To expand all of the rows and columns evenly, you give them each a weight of 1. Figure 9.14 shows the `circlearea` program without and with the expansion of the grid enabled.



[FIGURE 9.14] The `circlearea` GUI with row and column expansion

The methods **`rowconfigure`** and **`columnconfigure`** set the expansion weights on rows and columns, respectively. Each method expects two arguments. These are the number of the row or column and the weight.

Before setting the expansion weights of the rows and columns in the current frame, you must set these weights for the row and column of the current frame in the root window. After doing that, you grid the frame with the **sticky** attribute set to expand in four directions. Here is the code for expanding the frame within the root window:

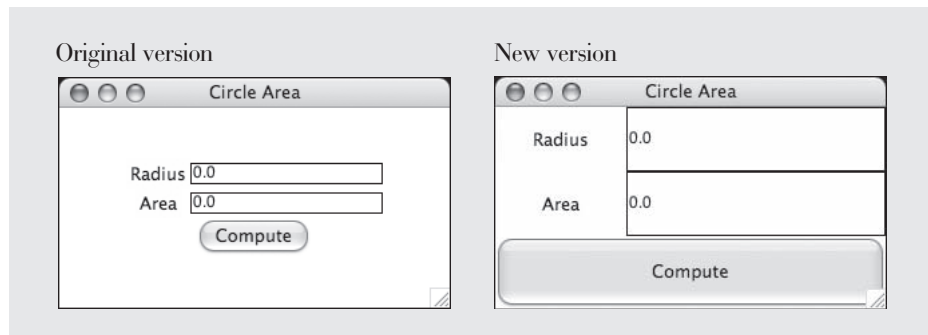
```
self.master.rowconfigure(0, weight = 1)
self.master.columnconfigure(0, weight = 1)
self.grid(sticky = W+E+N+S)
```

Finally, after the widgets have been positioned in their grid cells, you set the expansion weights of the current frame's three rows and two columns as follows:

```
for row in range(3):
    self.rowconfigure(row, weight = 1)
for column in range(2):
    self.columnconfigure(column, weight = 1)
```

If the widgets are centered within their cells, their positions will now depend on the window's current dimensions.

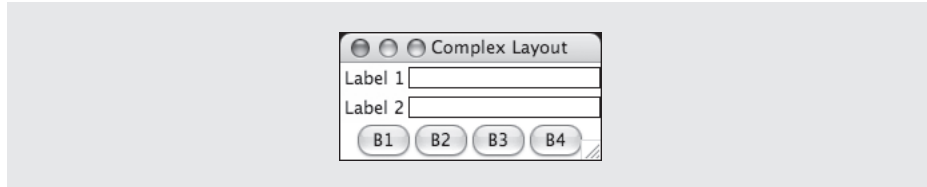
Finally, let's consider how to get widgets to expand within their cells. You assume that their rows and columns have been set to expand as well. Then you set the **sticky** attributes of the widgets' grid cells to the appropriate values. The value **W+E** expands horizontally, **N+S** expands vertically, and **W+E+N+S** expands in all four directions to fill the cell. Figure 9.15 shows the **circlearea** GUI before and after this type of expansion in all four directions is enabled.



[FIGURE 9.15] The **circlearea** GUI with widget expansion

9.4.6 Using Nested Frames to Organize Components

Suppose that a GUI requires a row of four command buttons beneath two columns of labels and entry fields, as shown in Figure 9.16.



[FIGURE 9.16] A complex grid layout

This grid appears to have two columns in two rows and four columns in a third row. It is difficult, but not impossible, to create this complex layout with a single grid. However, it would still take a great deal of extra work with the grid attributes to get the layout to look like the one in Figure 9.16.

A more natural design decomposes the window into two nested frames, each containing its own grid. The top frame contains a 2 by 2 grid of labels and entry fields, whereas the bottom frame contains a 1 by 4 grid of buttons. To code this design, a nested frame, sometimes called a **pane**, is instantiated with its parent frame as an argument. The new frame is then added to its parent's grid and becomes the parent of the widgets in its own grid. Here is the code for laying out the GUI shown in Figure 9.16:

```
class ComplexLayout(Frame):  
  
    def __init__(self):  
  
        # Create the main frame  
        Frame.__init__(self)  
        self.master.title("Complex Layout")  
        self.grid()  
  
        # Create the nested frame for the data pane  
        self._dataPane = Frame(self)  
        self._dataPane.grid(row = 0, column = 0)
```

continued

```

# Create and add widgets to the data pane
self._label1 = Label(self._dataPane, text = "Label 1")
self._label1.grid(row = 0, column = 0)
self._entry1 = Entry(self._dataPane)
self._entry1.grid(row = 0, column = 1)
self._label2 = Label(self._dataPane, text = "Label 2")
self._label2.grid(row = 1, column = 0)
self._entry2 = Entry(self._dataPane)
self._entry2.grid(row = 1, column = 1)

# Create the nested frame for the button pane
self._buttonPane = Frame(self)
self._buttonPane.grid(row = 1, column = 0)

# Create and add buttons to the button pane
self._button1 = Button(self._buttonPane, text = "B1",)
self._button2 = Button(self._buttonPane, text = "B2",)
self._button3 = Button(self._buttonPane, text = "B3",)
self._button4 = Button(self._buttonPane, text = "B4",)
self._button1.grid(row = 0, column = 0,)
self._button2.grid(row = 0, column = 1,)
self._button3.grid(row = 0, column = 2,)
self._button4.grid(row = 0, column = 3,)

```

9.4.7 Multi-Line Text Widgets

Entry fields support the input and output of a single line of text. Python includes a **Text** widget for the display of multiple lines of text. This component has a powerful range of features and operations, but in this subsection we restrict our discussion to simple output of text.

During instantiation, the programmer can specify the width in columns and the height in rows of the text that is initially visible in the **Text** widget. The widget's **wrap** attribute by default is **CHAR**, which wraps text to the next line when a character is about to go off the right boundary of the widget. The **wrap** attribute can be set to **WORD** for a more pleasing effect or to **NONE** for no wrapping.

There are various ways to allow a user to view text that extends beyond the visible area of a **Text** widget. The easiest way for the programmer is to allow the **Text** widget to expand with its grid cell, as shown earlier. However, this forces the user to expand the window to view the hidden text, and there is a limit to this expansion. Alternatively, scroll bars can be added to a **Text** widget to allow the user to scroll through the text. In this section, we examine the first alternative.

As with an **Entry** widget, a user's editing within a **Text** widget can be disabled by setting its **state** attribute to **DISABLED**. However, this attribute must be reset to **NORMAL** to send output to the **Text** widget.

Text within a **Text** widget is accessed by index positions. These positions are specified not as integers, but as strings. The general format of an index is

```
"rowNumber.characterNumber"
```

where **rowNumber** is counted from 1, and **characterNumber** is counted from 0. Thus, the index of the first character, if there is one, is **"1.0"**. The **tkinter** constant **END** represents the position following the last character in a **Text** widget.

The method **insert** is used to send a string to a **Text** widget. The method expects an index as its first argument and a string as its second argument. The method **insert** inserts the string at the position specified by the index. Thus, if we assume that **output** refers to a **Text** widget, the expression

```
output.insert("1.0", "Python rules!")
```

places the string before any existing text, whereas the expression

```
output.insert(END, "Python rules!")
```

places the string after any existing text. You can use expressions such as the last one when you want to append outputs to a **Text** widget.

You can use the method **delete** to clear a **Text** widget. This method is also index-based; as arguments it expects the beginning index and the index of the character after the string to be deleted from the widget. Thus, the following expression clears the widget **output** of all of its text:

```
output.delete("1.0", END)
```

When you want to reset the contents of a **Text** widget to a new string, rather than append this string to them, you can first delete the existing contents and then insert the new string. Our next demo program displays a 20 by 5 **Text** widget and two buttons that allow the user to test these options. The user can

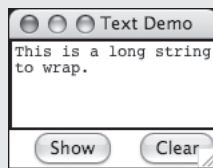
also edit the text within the **Text** widget. The GUI is shown in Figure 9.17. Here is the code:

```
from tkinter import *
class TextDemo(Frame):
    """Demonstrates a multi-line text area."""

    def __init__(self):
        """Sets up the window and widgets."""
        Frame.__init__(self)
        self.master.title("Text Demo")
        self.master.rowconfigure(0, weight = 1)
        self.master.columnconfigure(0, weight = 1)
        self.grid(sticky = W+E+N+S)
        self._text = "This is a long string to wrap."
        self._outputArea = Text(self,
                                width = 20,
                                height = 5,
                                wrap = WORD)
        self._outputArea.grid(row = 0, column = 0,
                              colspan = 2,
                              sticky = W+E+N+S)
        self._showButton = Button(self,
                                   text = "Show",
                                   command = self._show)
        self._showButton.grid(row = 1, column = 0)
        self._clearButton = Button(self,
                                    text = "Clear",
                                    command = self._clear)
        self._clearButton.grid(row = 1, column = 1)
        self.rowconfigure(0, weight = 1)
        self.columnconfigure(0, weight = 1)

    def _show(self):
        self._outputArea.insert("1.0", self._text)

    def _clear(self):
        self._outputArea.delete("1.0", END)
```



[FIGURE 9.17] A **Text** widget

9.4.8 Scrolling List Boxes

Lists of strings can be displayed in **list boxes**. The `tkinter.Listbox` class includes a wide array of methods for managing items in a list box. Some of these are listed in Table 9.5.

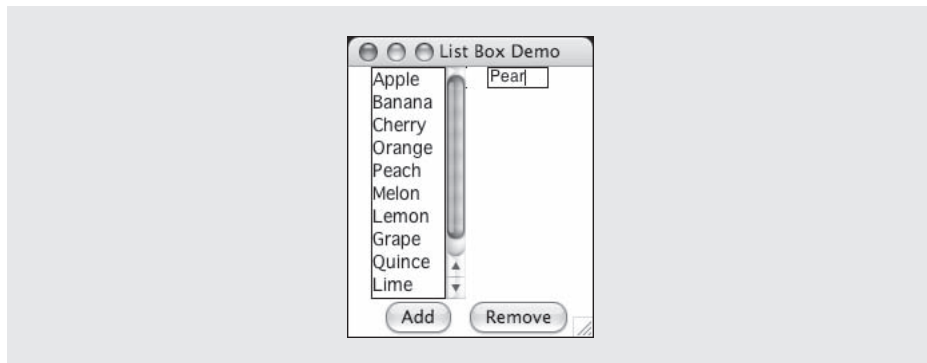
Listbox METHOD	WHAT IT DOES
<code>box.activate(index)</code>	Selects the string at index , counting from 0.
<code>box.curselection()</code>	Returns a tuple containing the currently selected index, if there is one, or the empty tuple.
<code>box.delete(index)</code>	Removes the string at index .
<code>box.get(index)</code>	Returns the string at index .
<code>box.insert(index, string)</code>	Inserts the string at index, shifting the remaining lines down by one position.
<code>box.see(index)</code>	Adjusts the position of the list box so the string at index is visible.
<code>box.size()</code>	Returns the number of strings in the list box.
<code>box.xview()</code>	Used with a horizontal scroll bar to effect scrolling.
<code>box.yview()</code>	Used with a vertical scroll bar to effect scrolling.

[TABLE 9.5] Some **Listbox** methods

Access to the items in a list box is index-based. The index of any item in a list box may be specified with a zero-based integer. The constants **ACTIVE** and **END** also specify index positions in a list box. The currently selected item is located at the **ACTIVE** index, whereas the end of the list is at the **END** index. The default **selectmode** attribute of a list box is **MULTIPLE**, meaning that many items can be selected at once. This attribute can be reset to **SINGLE**. As with **Text** widgets, you can specify the width and height of a list box in columns and rows.

Long lists of items typically extend beyond the visible height of a list box. You can accommodate the user's need to see them by allowing the list box to expand vertically, but a much more convenient method is to associate a **scroll bar** with the list box. The user moves the list of items under the visible area of the list box by dragging this bar up and down with the mouse. The `tkinter.Scrollbar` class supports this mechanism.

Let us develop a program that illustrates a scrolling list box. The GUI for this program, shown in Figure 9.18, displays several strings in a scrolling list box at start-up. The entry field on the right is for the input of new strings. The user can add these to the end of the list by selecting the **Add** button. The user can remove the currently selected string by selecting the **Remove** button. The user selects a string by clicking it with the mouse.



[FIGURE 9.18] A scrolling list box

We now highlight important steps in the code for this program. You begin by creating a nested frame named `_listPane` to hold the list box and its scroll bar. This frame is set to expand vertically within its grid cell:

```
self._listPane = Frame(self)
self._listPane.grid(row = 0, column = 0, sticky = N+S)
```

You then create the scroll bar and grid it within its parent widget, the list pane. A scroll bar can have either a vertical or a horizontal orientation. Its **orient** attribute is here set to **VERTICAL**:

```
self._yScroll = Scrollbar(self._listPane,
                          orient = VERTICAL)
self._yScroll.grid(row = 0, column = 1, sticky = N+S)
```

The list box is then instantiated and placed in the list pane as well. Its **yscrollcommand** attribute is set to the scroll bar's **set** method. The scroll bar's

command attribute is set to the list box's **yview** method. These two methods collaborate to scroll the items in the list box when the user drags the scroll bar:

```
self._theList = Listbox(self._listPane,
                        width = 6,
                        height = 10,
                        selectmode = SINGLE,
                        yscrollcommand = self._yScroll.set)
self._theList.grid(row = 0, column = 0, sticky = N+S)
self._yScroll["command"] = self._theList.yview
```

Several items are added to the list box, and the first one is made active:

```
self._theList.insert(END, "Apple")
self._theList.insert(END, "Banana")
self._theList.insert(END, "Cherry")
self._theList.insert(END, "Orange")
self._theList.activate(0)
```

Finally, both the main frame's first row and the nested frame's row are configured to expand vertically:

```
self.rowconfigure(0, weight = 1)
self._listPane.rowconfigure(0, weight = 1)
```

The method to add items to the list box places them at the end of the items currently there:

```
def _add(self):
    """If an input is present, insert it at the
    end of the items in the list box and scroll to it."""
    item = self._inputVar.get()
    if item != "":
        self._theList.insert(END, item)
        self._theList.see(END)
```

The method to remove items from the list box relies on the index of the selected item:

```
def _remove(self):
    """If there are items in the list, remove
    the selected item."""
    if self._theList.size() > 0:
        self._theList.delete(ACTIVE)
```

9.4.9

Mouse Events

To a large extent, a user interacts with a GUI-based program by manipulating widgets with the mouse. A hidden, event-driven loop automatically detects different types of mouse events, such as button presses, button releases, and mouse dragging, and triggers any corresponding event-handling methods that have been defined in the program. We have exploited this mechanism to respond to clicks on command buttons in many of our examples. However, the programmer can associate methods with any mouse events that occur in any widget. Table 9.6 lists the different types of mouse events that can occur.

TYPE OF MOUSE EVENT	DESCRIPTION
<ButtonPress- <i>n</i> >	Mouse button <i>n</i> has been pressed while the mouse cursor is over the widget; <i>n</i> can be 1 (left button), 2 (middle button), or 3 (right button).
<ButtonRelease- <i>n</i> >	Mouse button <i>n</i> has been released while the mouse cursor is over the widget; <i>n</i> can be 1 (left button), 2 (middle button), or 3 (right button).
<B <i>n</i> -Motion>	The mouse is moved with button <i>n</i> held down.
<Prefix-Button- <i>n</i> >	The mouse has been clicked over the widget; <i>Prefix</i> can be Double or Triple .
<Enter>	The mouse cursor has entered the widget.
<Leave>	The mouse cursor has left the widget.

[TABLE 9.6] Mouse events

You can associate a mouse event and an event-handling method with a widget by calling the **bind** method. This method expects a string containing one of the

mouse events listed in Table 9.6 as its first argument, and the method to be triggered as its second argument.

For example, suppose the list box demo discussed earlier should respond by displaying a list item in the entry field when it is selected in the list box. The selection is finished when the mouse is released after pressing an item. Let's assume that a method named `_get` should be triggered when this happens. Then the code for binding this method to that event for the list box is the following:

```
self._theList.bind("<ButtonRelease-1>", self._get)
```

Now all you have to do is define the `_get` method. This method has a single parameter named `event`. This parameter will automatically be bound to the event object that triggered the method. The method `_get` does nothing if the list box is empty. Otherwise, it fetches the index of the currently selected item and uses it to fetch the current item itself. This string is then sent to the container variable for the entry field. Here is the code for the `_get` method:

```
def _get(self, event):
    """If the list is not empty, copy the selected
    string to the entry field."""
    if self._theList.size() > 0:
        index = self._theList.curselection()[0]
        self._inputVar.set(self._theList.get(index))
```

9.4.10 Keyboard Events

GUI-based programs can also respond to various keyboard events. Table 9.7 lists some commonly occurring ones.

TYPE OF KEYBOARD EVENT	DESCRIPTION
<code><KeyPress></code>	Any key has been pressed.
<code><KeyRelease></code>	Any key has been released.
<code><KeyPress-key></code>	<i>key</i> has been pressed.
<code><KeyRelease-key></code>	<i>key</i> has been released.

[TABLE 9.7] Some key events

As with mouse events, you can associate any key events with widgets in such a manner that the events trigger methods. Perhaps the most common event is pressing the return key when the mouse cursor has become the insertion point in an entry field. This event might signal the end of an input and a request for processing.

Key events and their handlers are associated with a widget by using the **bind** method discussed earlier. Let's revisit the circle area program to allow the user to compute the area by pressing the return key while the insertion point is in the radius field.

You bind the key press event to a handler for the **_radiusEntry** widget as follows:

```
self._radiusEntry.bind("<KeyPress-Return>",  
                        lambda event: self._area())
```

You cannot use the **_area** method directly as the event handler, because **_area** does not have a parameter for the event. Instead, you package a call of **_area** within a **lambda** function that accepts the event and ignores it.

9.4 Exercises

- 1 Write a code segment that centers the labels RED, WHITE, and BLUE vertically in a GUI window. The text of each label should have the color that it names, and the window's background color should be green. The background color of each label should also be black.
- 2 Write a code segment that centers the labels COURIER, HELVETICA, and TIMES horizontally in a GUI window. The text of each label should have the type font family that it names. Substitute a different font if necessary.
- 3 Write a code segment that uses a loop to create and place nine buttons into a 3 by 3 grid. Each button should be labeled with a number, starting with 1 and increasing across each row.
- 4 Describe how a vertical scroll bar is associated with a list box.

Summary

- GUI-based programs display information using graphical components in a window. They allow a user to manipulate information by manipulating GUI components with a mouse.
- A GUI-based program responds to user events by running methods to perform various tasks.
- The model/view/controller pattern assigns the roles and responsibilities in a GUI-based program to three different sets of classes. The view is responsible for displaying data and receiving user inputs. The model is responsible for managing the program's data. The controller is responsible for handling the communications between the model and the view.
- The **tkinter** module includes classes, functions, and constants used in GUI programming.
- A GUI-based program is structured as a main window class. This class extends the **Frame** class. The **__init__** method in the main window class creates and lays out the window objects. The main window class also includes the definitions of any event-handling methods.
- Examples of window components are labels (either text or images), command buttons, entry fields, multi-line text areas, and list boxes.
- Pop-up dialog boxes are used to display messages and ask Yes/No questions. Functions for these are included in the **tkinter.messagebox** module.
- Window objects can be arranged in a window under the influence of a grid layout. The grid's attributes can be set to allow components to expand or align in any direction.
- Complex layouts can be decomposed into several panes of components.
- Each component has attributes for the foreground color and background color. Colors are represented using the RGB system in hexadecimal format.
- Text has a type font attribute that allows the programmer to specify the family, size, and other attributes of a font.
- The **command** attribute of a button can be set to a method that handles a button click.
- Mouse and keyboard events can be associated with handler methods for window objects by using the **bind** method.

REVIEW QUESTIONS

- 1 In contrast to a terminal-based program, a GUI-based program
 - a completely controls the order in which the user enters inputs
 - b can allow the user to enter inputs in any order
- 2 The main window class in a GUI-based program is a subclass of
 - a **Text**
 - b **Frame**
 - c **Window**
- 3 The attribute used to attach an event-handling method to a button is named
 - a **pressevent**
 - b **onclick**
 - c **command**
- 4 The model classes are responsible for
 - a managing a program's data
 - b displaying a program's data
- 5 The controller methods
 - a are triggered when events occur in the view
 - b manage a program's data
 - c display a program's data
- 6 The window component that allows a user to move the text visible beneath a **Text** widget is a
 - a list box
 - b label
 - c scroll bar
- 7 The **sticky** attribute
 - a controls the alignment of a window component in its grid cell
 - b makes it difficult for a window component to be moved
- 8 The field used to set frame attributes is called
 - a **master**
 - b **mister**

- 9 Generally speaking, it is better to
 - a define a main window of a fixed size
 - b allow the user to alter the size of a main window
- 10 The rows and columns in a grid layout are numbered starting from
 - a (0, 0)
 - b (1, 1)

PROJECTS

- 1 Write a GUI-based program that implements the **bouncy** program example discussed in Section 9.1.
- 2 Write a GUI-based program that allows the user to convert temperature values between degrees Fahrenheit and degrees Celsius. The interface should have labeled entry fields for these two values. These components should be arranged in a grid where the labels occupy the first row and the corresponding fields occupy the second row. At start-up, the Fahrenheit field should contain 32.0, and the Celsius field should contain 0.0. The third row in the window contains two command buttons, labeled >>>> and <<<<. When the user presses the first button, the program should use the data in the Fahrenheit field to compute the Celsius value, which should then be output to the Celsius field. The second button should perform the inverse function.
- 3 A terminal-based program that uses Newton's method to compute square roots is described in Chapter 3. Recast this program as a GUI-based program. The user should be able to view successive approximations by clicking a command button.

The interface should have two labeled entry fields, one for the input number and the other for the output of the square root. The interface should include two command buttons. A button labeled **Estimate** should compute and display the next guess based on the previous one. A button labeled **Reset** should set the input and output fields to 0.0. At start-up and after each reset, the program's initial guess should be 0.0. If the program's initial guess is 0.0 and the user's input is greater than 0.0, the program's first guess should be set to the input divided by 2.0. Otherwise, the program's new guess should be set using Newton's approximation formula.

- 4 Write a GUI-based program that plays the game of Blackjack as described in Chapter 8. The window should display images of the player's cards and dealer's cards as they are drawn. The window should include the command buttons **Hit**, **Pass**, and **New Game**, and a status field to display the game's outcome.
- 5 Write a GUI-based program that allows a bank manager to view and manipulate the accounts in a bank. The window should display the information for the currently selected account in editable entry fields. Command buttons should allow the user to navigate to the next account and the previous account, assuming that the accounts are ordered by a PIN. Add a method `getPins()` to the **Bank** class. This method should build and return a sorted list of the PINs in the bank. The GUI should use this method to help locate the first account, next account, and previous account. Command buttons should also allow the user to remove an account, save an account's edited information, and add a new account. When a new account is added, the entry fields should be reset to default values, and the **Update Account** button should create the account.
- 6 The TidBit Computer Store (Chapter 3, Project 10) has a credit plan for computer purchases. Inputs are the annual interest rate and the purchase price. Monthly payments are 5 percent of the listed purchase price, minus the down payment, which must be 10 percent of the purchase price. Write a GUI-based program that displays labeled fields for the inputs and a text area for the output. The program should display a table, with appropriate headers, of a payment schedule for the lifetime of the loan. Each row of the table should contain the following items:
 - The month number (beginning with 1)
 - The current total balance owed
 - The interest owed for that month
 - The amount of principal owed for that month
 - The payment for that month
 - The balance remaining after payment

The amount of interest for a month is equal to $\text{balance} * \text{rate} / 12$. The amount of principal for a month is equal to the monthly payment minus the interest owed.

Your program should include separate classes for the model and the view. The model should include a method that expects the two inputs as arguments and returns a formatted string for output by the view.

- 7 Write a GUI-based program that simulates a simple pocket calculator. The GUI displays a single entry field for output. The GUI should also display a keypad of buttons for the 10 digits and 6 command buttons labeled **+**, **-**, *****, **/**, **C**, and **=**. The command **C** should clear the output field. The command **=** calculates an answer and displays it in the field. The program should build a string from the user's button clicks and echo it in the field. The program should detect any errors during this process and display the word "ERR" in the field.
- 8 Write a GUI-based program that allows the user to open, edit, and save text files. The GUI should include a labeled entry field for the filename and a multi-line text widget for the text of the file. The user should be able to scroll through the text by manipulating a vertical scrollbar. Include command buttons labeled **Open**, **Save**, and **New** that allow the user to open, save, and create new files. The **New** command should then clear the text widget and the entry widget.
- 9 Write a GUI-based program that implements an image browser for your computer's hard disk. At start-up, the program should load a scrolling list box with three types of items:
 - The filenames of the images in the current working directory
 - The names of any subdirectories within the current working directory
 - The string **".."**

The pathname of the current working directory is also displayed in an entry field. When the user selects an item in the list box and presses the **Go** button, one of three things can happen:

- If the item is an image filename, the image is loaded and displayed.
- If the item is a subdirectory, the program attaches to that directory and refreshes the list box with its contents.
- If the item is the string **".."**, the program attaches to the parent directory if there is one and refreshes the list box with its contents.

In the last two cases, the contents of the entry field are also updated.

- 10 Write a GUI-based program that allows the user to play a game of tic-tac-toe with the computer. The main window should display a 3 by 3 grid of empty buttons. When the user presses an empty button, an **X** should appear. The computer should then respond by checking for a winner, and then placing an **O** on an empty button if there is no winner. The computer should then check for a winner again. A **Reset** button should reset the game and the window to their initial state. Allow the computer to place its mark on a randomly chosen button.

[CHAPTER] 10

MULTITHREADING, NETWORKS, AND Client/Server Programming

After completing this chapter, you will be able to:

- Describe what threads do and how they are manipulated in an application
- Code an algorithm to run as a thread
- Use conditions to solve a simple synchronization problem with threads
- Use IP addresses, ports, and sockets to create a simple client/server application on a network
- Decompose a server application with threads to handle client requests efficiently
- Restructure existing applications for deployment as client/server applications on a network

Thus far in this book, we have explored ways of solving problems by using multiple cooperating algorithms and data structures. Another commonly used strategy for problem solving involves the use of multiple threads. Threads describe processes that can run concurrently to solve a problem. They can also be organized in a system of clients and servers. For example, a Web browser runs in a client thread and allows a user to view Web pages that are sent by a Web server, which runs in a server thread. Client and server threads can run concurrently on a single computer or can be distributed across several computers that are linked in a network. The technique of using multiple threads in a program is known as multithreading. This chapter offers an introduction to multithreading, networks, and client/server programming. We provide just enough material to get you started with these topics; more complete surveys are available in advanced computer science courses.

You are well aware that an algorithm describes a computational process that runs to completion. You are also aware that a process consumes resources, such as CPU cycles and memory. Until now, we have associated an algorithm or a program with a single process, and we have assumed that this process runs on a single computer. However, your program's process is not the only one that runs on your computer, and a single program could describe several processes that could run concurrently on your computer or on several networked computers. The following historical summary shows how this is the case.

Time-sharing systems: In the late 1950s and early 1960s, computer scientists developed the first time-sharing operating systems. These systems allowed several programs to run concurrently on a single computer. Instead of giving their programs to a human scheduler to run one after the other on a single machine, users logged in to the computer via remote terminals. They then ran their programs and had the illusion, if the system performed well, of having sole possession of the machine's resources (CPU, disk drives, printer, etc.). Behind the scenes, the operating system created processes for these programs, gave each process a turn at the CPU and other resources, and performed all the work of scheduling, saving state during context switches, and so forth. Time-sharing systems are still in widespread use in the form of Web servers, e-mail servers, print servers, and other kinds of servers on networked systems.

Multiprocessing systems: Most time-sharing systems allow a single user to run one program and then return to the operating system to run another program before the first program is finished. The concept of a single user running several programs at once was extended to desktop microcomputers in the late 1980s, when these machines became more powerful. For example, the Macintosh MultiFinder allowed a user to run a word processor, a spreadsheet, and the Finder (the file browser) concurrently and to switch from one application to another by selecting an application's window. Users of stand-alone PCs now take this capability for granted. A related development was the ability of a program to start another program by "forking," or creating a new process. For example, a word processor might create another process to print a document in the background, while the user is staring at the window thinking about the next words to type.

Networked or distributed systems: The late 1980s and early 1990s saw the rise of networked systems. At that time, the processes associated with a single program or with several programs began to be distributed across several CPUs linked by high-speed communication lines. Thus, for example, the Web browser that appears to be running on my machine is actually making requests as a client to a Web server application that runs on a multiuser machine at the local Internet

service provider. The problems of scheduling and running processes are more complex on a networked system, but the basic ideas are the same.

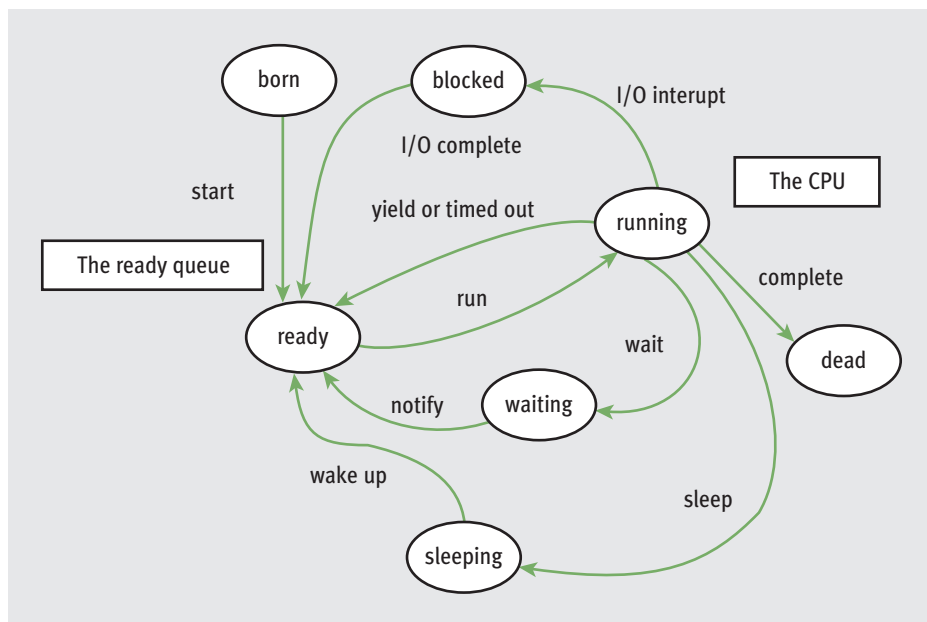
Parallel systems: As CPUs became less expensive and smaller, it became feasible to run a single program on several CPUs at once. **Parallel computing** is the discipline of building the hardware architectures, operating systems, and specialized algorithms for running a program on a cluster of processors. The multi-core technology now found in most new PCs can be used to run a single program or multiple programs on several processors simultaneously.

10.1.1 Threads

Whether networked or stand-alone machines, most modern computers use threads to represent processes. For example, a Web browser uses one thread to load an image from the Internet while using another thread to format and display text. The Python Virtual Machine runs several threads that you have already used without realizing it. For example, the IDLE editor runs as a separate thread, as does your main Python application program.

In Python, a thread is an object like any other in that it can hold data, be run with methods, be stored in data structures, and be passed as parameters to methods. However, a thread can also be executed as a process. Before it can execute, a thread's class must implement a **run** method.

During its lifetime, a thread can be in various states. Figure 10.1 shows some of the states in the lifetime of a Python thread. In this diagram, the box labeled “The ready queue” is a data structure, whereas the box labeled “The CPU” is a hardware resource. The thread states are the labeled ovals.



[FIGURE 10.1] States in the life of a thread

After it is created, a thread remains inactive until someone runs its **start** method. Running this method also makes the thread “ready” and places a reference to it in the **ready queue**. A queue is a data structure that enforces first-come, first-served access to a single resource. The resource in this case is the CPU, which can execute the instructions of just one thread at a time. A newly started thread’s **run** method is also activated. However, before its first instruction can be executed, the thread must wait its turn in the ready queue for access to the CPU. After the thread gets access to the CPU and executes some instructions in its **run** method, the thread can lose access to the CPU in several ways:

- **Time-out**—Most computers running Python programs automatically time-out a running thread every few milliseconds. The process of automatically timing-out, also known as **time slicing**, has the effect of pausing the running thread’s execution and sending it to the rear of the ready queue. The thread at the front of the ready queue is then given access to the CPU.
- **Sleep**—A thread can be put to sleep for a given number of milliseconds. When the thread wakes up, it goes to the rear of the ready queue.

- **Block**—A thread can wait for some event, such as user input, to occur. When a blocked thread is notified that an event has occurred, it goes to the rear of the ready queue.
- **Wait**—A thread can voluntarily relinquish the CPU to wait for some condition to become true. A waiting thread can be notified when the condition becomes true and move again to the rear of the ready queue.

When a thread gives up the CPU, the computer saves its state, so that when the thread returns to the CPU, its **run** method can pick up where it left off. The process of saving or restoring a thread's state is called a **context switch**.

When a thread's **run** method has executed its last instruction, the thread dies as a process but continues to exist as an object. A thread object can also die if it raises an exception that is not handled.

Python's **threading** module includes resources for creating threads and managing multithreaded applications. The most common way to create a thread is to define a class that extends the class **threading.Thread**. The new class should include a **run** method that executes the algorithm in the new thread. The **start** method places a thread at the rear of the ready queue. The next code segment defines a simple thread class that prints its name. The session that follows instantiates this class and starts up the thread.

```
from threading import Thread

class MyThread(Thread):

    def __init__(self):
        Thread.__init__(self, name = "My Thread")

    def run(self):
        print("Hello, my name is %s" % self.getName())

>>> process = MyThread()
>>> process.start()
Hello, my name is My Thread
>>>
```

Note that the thread's **run** method is invoked automatically by **start**. The **Thread** class maintains an instance variable for the thread's name and includes the associated methods **getName** and **setName**. Table 10.1 lists some important **Thread** methods.

Thread METHOD	WHAT IT DOES
<code>__init__(name = None)</code>	Initializes the thread's name.
<code>getName()</code>	Returns the thread's name.
<code>setName(newName)</code>	Sets the thread's name to newName .
<code>run()</code>	Executes when the thread acquires the CPU.
<code>start()</code>	Makes the new thread ready. Raises an exception if run more than once.
<code>isAlive()</code>	Returns True if the thread is alive or False otherwise.

[TABLE 10.1] Some **Thread** Methods

Other important resources used with threads include the function **time.sleep** and the class **threading.Condition**. We now consider some example programs that illustrate the behavior of these resources.

10.1.2 Sleeping Threads

In our first example, we develop a program that allows the user to start several threads. Each thread does not do much when started; it simply prints a message, goes to sleep for a random number of seconds, and then prints a message and terminates on waking up. The program allows the user to specify the number of threads to run and the maximum sleep time. When a thread is started, it prints a message identifying itself and its sleep time and then goes to sleep. When a thread wakes up, it prints another message identifying itself. A session with this program is shown in Figure 10.2. Note that the Python program is launched from a terminal prompt rather than from an IDLE window. Because IDLE itself runs in a thread, it is not generally a good idea to test a multithreaded application in that environment.

```
% python sleepythreads.py
Enter the number of threads: 3
Enter the maximum sleep time: 6
Thread 1, sleep interval: 3 seconds
Thread 2, sleep interval: 3 seconds
Thread 3, sleep interval: 1 second
Thread 3 waking up
Thread 1 waking up
Thread 2 waking up
```

[FIGURE 10.2] A run of the sleeping threads program

The following points can be concluded from the example in Figure 10.2:

- When a thread goes to sleep, the next thread has an opportunity to acquire the CPU and display its information in the view.
- The threads do not necessarily wake up in the order in which they were started. The size of the sleep interval determines this order. In Figure 10.2, thread 3 has the shortest sleep time, so it wakes up first. Thread 1 wakes up before thread 2 because their sleep intervals are the same, and 1 is started before 2.

The program consists of the class **SleepyThread**, a subclass of **Thread**, and a **main** function. When called within a thread's **run** method, the function **time.sleep** puts that thread to sleep for the specified number of seconds. Here is the code:

```
"""
File: sleepythreads.py

Illustrates concurrency with multiple threads.
"""

import random, time
from threading import Thread

class SleepyThread(Thread):
    """Represents a sleepy thread."""

    def __init__(self, number, sleepMax):
        """Create a thread with the given name
        and a random sleep interval less than the maximum. """
        Thread.__init__(self, name = "Thread " + str(number))
        self._sleepInterval = random.randint(1, sleepMax)

    def run(self):
```

continued


```

        """Print the thread's name and sleep interval and sleep
        for that interval. Print the name again at wake-up. """
        print("%s, sleep interval: %d seconds" % \
              (self.getName(), self._sleepInterval))
        time.sleep(self._sleepInterval)
        print("%s waking up" % self.getName())

def main():
    """Create the user's number of threads with sleep
    intervals less than the user's maximum. Then start
    the threads"""
    numThreads = int(input("Enter the number of threads: "))
    sleepMax = int(input("Enter the maximum sleep time: "))
    threadList = []
    for count in range(numThreads):
        threadList.append(SleepyThread(count + 1, sleepMax))
    for thread in threadList: thread.start()

main()

```

10.1.3 Producer, Consumer, and Synchronization

In the previous example, the threads ran independently and did not interact. However, in many applications, threads interact by sharing data. Threads that interact by sharing data are said to have a **producer/consumer relationship**. Think of an assembly line in a factory. Worker A, at the beginning of the line, produces an item that is then ready for access by the next person on the line, Worker B. In this case, Worker A is the producer, and Worker B is the consumer. Worker B then becomes the producer, processing the item in some way until it is ready for Worker C, and so on.

Three requirements must be met for the assembly line to function properly:

- 1 A producer must produce each item before a consumer consumes it.
- 2 Each item must be consumed before the producer produces the next item.
- 3 A consumer must consume each item just once.

Let us now consider a computer simulation of the producer/consumer relationship. In its simplest form, the relationship has only two threads: a producer and a consumer. They share a single data cell that contains an integer. The producer sleeps for a random interval, writes an integer to the shared cell, and generates the next integer to be written, until the integer reaches an upper bound. The consumer sleeps for a random interval and reads the integer from the shared cell, until the

integer reaches the upper bound. Figure 10.3 shows two runs of this program. The user enters the number of accesses (data items produced and consumed). The output announces that the producer and consumer threads have started up and shows when each thread accesses the shared data.

Enter the number of accesses: 4	Enter the number of accesses: 4
Starting the threads	Starting the threads
Producer starting up	Producer starting up
Consumer starting up	Consumer starting up
Producer setting data to 1	Consumer accessing data -1
Consumer accessing data 1	Producer setting data to 1
Producer setting data to 2	Producer setting data to 2
Consumer accessing data 2	Consumer accessing data 2
Producer setting data to 3	Consumer accessing data 2
Consumer accessing data 3	Producer setting data to 3
Producer setting data to 4	Consumer accessing data 3
Producer is done producing	Consumer is done consuming
Consumer accessing data 4	Producer setting data to 4
Consumer is done consuming	Producer is done producing

[FIGURE 10.3] Two runs of the producer/consumer program

Some bad things happen in the second run of the program (lines in boldface type on the right of Figure 10.3):

- 1 The consumer accesses the shared cell before the producer has written its first datum.
- 2 The producer then writes two consecutive data (1 and 2) before the consumer has accessed the cell again.
- 3 The consumer accesses data 2 twice.
- 4 The producer then writes data 4 after the consumer is finished.

The producer produces all of its data as expected, but the consumer can access data that are not there, can miss data, and can access the same data more than once. These are known as **synchronization problems**. Before we explain why they occur, we present the essential parts of the program itself, which consists of the four resources in Table 10.2.

CLASS OR FUNCTION	ROLE AND RESPONSIBILITY
main	Manages the user interface. Creates the shared cell and producer and consumer threads and starts the threads.
SharedCell	Represents the shared data, which is an integer (initially -1).
Producer	Represents the producer process. Repeatedly writes an integer to the cell and increments the integer, until it reaches an upper bound.
Consumer	Represents the consumer process. Repeatedly reads an integer from the cell, until it reaches an upper bound.

[TABLE 10.2] The classes and **main** function in the producer/consumer program

The code for the **main** function is similar to the one in the previous example:

```
def main():
    """Get the number of accesses from the user,
    create a shared cell, and create and start up
    a producer and a consumer."""
    accessCount = int(input("Enter the number of accesses: "))
    sleepMax = 4
    cell = SharedCell()
    producer = Producer(cell, accessCount, sleepMax)
    consumer = Consumer(cell, accessCount, sleepMax)
    print("Starting the threads")
    producer.start()
    consumer.start()
```

Here is the code for the classes **SharedCell**, **Producer**, and **Consumer**:

```
import time, random
from threading import Thread, currentThread

class SharedCell(object):
    """Shared data for the producer/consumer problem."""

    def __init__(self):
        self._data = -1
```

continued

```

def setData(self, data):
    """Producer's method to write to shared data."""
    print("%s setting data to %d" % \
          (currentThread().getName(), data))
    self._data = data

def getData(self):
    """Consumer's method to read from shared data."""
    print("%s accessing data %d" % \
          (currentThread().getName(), self._data))
    return self._data

class Producer(Thread):
    """Represents a producer."""

    def __init__(self, cell, accessCount, sleepMax):
        """Create a producer with the given shared cell,
        number of accesses, and maximum sleep interval."""
        Thread.__init__(self, name = "Producer")
        self._accessCount = accessCount
        self._cell = cell
        self._sleepMax = sleepMax

    def run(self):
        """Announce start-up, sleep, and write to shared cell
        the given number of times, and announce completion."""
        print("%s starting up" % self.getName())
        for count in range(self._accessCount):
            time.sleep(random.randint(1, self._sleepMax))
            self._cell.setData(count + 1)
            print("%s is done producing" % self.getName())

class Consumer(Thread):
    """Represents a consumer."""

    def __init__(self, cell, accessCount, sleepMax):
        """Create a producer with the given shared cell,
        number of accesses, and maximum sleep interval."""
        Thread.__init__(self, name = "Consumer")
        self._accessCount = accessCount
        self._cell = cell
        self._sleepMax = sleepMax

    def run(self):
        """Announce start-up, sleep, and read from shared cell
        the given number of times, and announce completion."""
        print("%s starting up" % self.getName())
        for count in range(self._accessCount):
            time.sleep(random.randint(1, self._sleepMax))
            value = self._cell.getData()
            print("%s is done consuming" % self.getName())

```

The cause of the synchronization problems is not hard to spot in this code. On each pass through their main loops, the threads sleep for a random interval of time. Thus, if the consumer thread has a shorter interval than the producer thread on a given cycle, the consumer wakes up sooner and accesses the shared cell before the producer has a chance to write the next datum. Conversely, if the producer thread wakes up sooner, it accesses the shared data and writes the next datum before the consumer has a chance to read the previous datum.

To solve this problem, we need to synchronize the actions of the producer and consumer threads. In addition to holding data, the shared cell must be in one of two states: writeable or not writeable. The cell is writeable if it has not yet been written to (at start-up) or if it has just been read from. The cell is not writeable if it has just been written to. These two conditions can now control the callers of the **setData** and **getData** methods in the **SharedCell** class as follows:

- 1 While the cell is writeable, the caller of **getData** (the consumer) must wait or suspend activity, until the producer writes a datum. When this happens, the cell becomes not writeable, the caller of **getData** is notified to resume activity, and the data are returned (to the consumer).
- 2 While the cell is not writeable, the caller of **setData** (the producer) must wait or suspend activity, until the consumer reads a datum. When this happens, the cell becomes writeable, the caller of **setData** is notified to resume activity, and the data are modified (by the producer).

To implement these restrictions, the **SharedCell** class now includes two additional instance variables:

- 1 A Boolean flag named **_writeable**. If this flag is **True**, only writing to the cell is allowed; if it is **False**, only reading from the cell is allowed.
- 2 An instance of the **threading.Condition** class. This object allows each thread to block until the Boolean flag is in the appropriate state to write to or read from the cell.

A **Condition** object is used to maintain a **lock** on a resource. When a thread acquires this lock, no other thread can access the resource, even if the acquiring thread is timed-out. After a thread successfully acquires the resource, it can do its work or relinquish the lock in one of two ways:

- 1 By calling the condition's **wait** method. This method causes the thread to block until it is notified that it can continue its work.
- 2 By calling the condition's **release** method. This method unlocks the resource and allows it to be acquired by other threads.

When other threads attempt to acquire a locked resource, they block until the thread is released or a thread holding the lock calls the condition's **notify** method. To summarize, the pattern for a thread accessing a resource with a lock is the following:

- Run **acquire** on the condition.
- While it's not OK to do the work
 - Run **wait** on the condition.
- Do the work with the resource.
- Run **notify** on the condition.
- Run **release** on the condition.

Table 10.3 lists the methods of the **Condition** class.

Condition METHOD	WHAT IT DOES
acquire()	Attempts to acquire the lock. Blocks if the lock is already taken.
release()	Relinquishes the lock, leaving it to be acquired by others.
wait()	Releases the lock, blocks the current thread until another thread calls notify or notifyAll on the same condition, and then reacquires the lock. If multiple threads are waiting, the notify method wakes up only one of the threads, while notifyAll always wakes up all of the threads.
notify()	Lets the next thread waiting on the lock know that it's available.
notifyAll()	Lets all threads waiting on the lock know that it's available.

[TABLE 10.3] The methods of the **Condition** class

Here is the code that shows the changes to the class **SharedCell**:

```
import time, random
from threading import Thread, currentThread, Condition

class SharedCell(object):
    """Shared data for the producer/consumer problem."""

    def __init__(self):
        self._data = -1
        self._writeable = True
        self._condition = Condition()

    def setData(self, data):
        """Producer's method to write to shared data."""
        self._condition.acquire()
        while not self._writeable:
            self._condition.wait()
        print("%s setting data to %d" % \
              (currentThread().getName(), data))
        self._data = data
        self._writeable = False
        self._condition.notify()
        self._condition.release()

    def getData(self):
        """Consumer's method to read from shared data."""
        self._condition.acquire()
        while self._writeable:
            self._condition.wait()
        print("%s accessing data %d" % \
              (currentThread().getName(), self._data))
        self._writeable = True
        self._condition.notify()
        self._condition.release()
        return self._data
```

We have only scratched the surface of the kinds of problems that can arise when programs run several threads. For example, the producer/consumer problem can involve multiple producers and/or consumers.

10.1

Exercises

- 1 What does a thread's **run** method do?
- 2 What is time slicing?
- 3 What is a synchronization problem?
- 4 What is the difference between a sleeping thread and a waiting thread?
- 5 Discuss how one might solve a producer/consumer problem with one producer and many consumers. You may assume that all of the consumers must consume each of the data values produced.
- 6 Assume that a producer and a consumer have access to a shared list of data. The producer's role is to replace the data value at each position, whereas the consumer simply accesses the replaced value, that is, the producer must replace before any consumer accesses. Describe how you would synchronize the producer and consumer so that they each can process the entire list.

10.2

Networks, Clients, and Servers

Clients and servers are applications or processes that can run locally on a single computer or remotely across a network of computers. As explained in the following sections, the resources required for this type of application are IP addresses, sockets, and threads.

10.2.1

IP Addresses

Every computer on a network has a unique identifier called an **IP address** (IP stands for Internet Protocol). This address can be specified either as an **IP number** or as an **IP name**. An IP number typically has the form *ddd.ddd.ddd.ddd*, where each *d* is a digit. The number of digits to the right or the left of a decimal point may vary but does not exceed three. For example, the IP number of the author's office computer might be 137.112.194.77. Because IP numbers can be difficult to remember, people customarily use an IP name to specify an IP address. For example, the IP name of the author's computer might be **lambertk**.

Python’s **socket** module includes two functions that can look up these items of information. These functions are listed in Table 10.4, followed by a short session showing their use.

socket FUNCTION	WHAT IT DOES
gethostname()	Returns the IP name of the host computer running the Python interpreter. Raises an exception if the computer does not have an IP address.
gethostbyname(ipName)	Returns the IP number of the computer whose IP name is ipName . Raises an exception if ipName cannot be found.

[TABLE 10.4] **socket** functions for IP addresses

```
>>> from socket import *
>>> gethostname()
'kenneth-lamberts-powerbook-g4-15.local'
>>> gethostbyname(gethostname())
'192.168.1.109'
>>> gethostbyname('Ken')

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    gethostbyname('Ken')
gaierror: (7, 'No address associated with nodename')
>>>
```

Note that these functions raise exceptions if they cannot locate the information. To handle this problem, one can embed these function calls in a **try-except** statement. First introduced in Chapter 8, this statement has the following general form:

```
try:
    <statements>
except Exception as exception:
    <statements>
```

The next code segment recovers from an unknown IP address error by printing the exception's error message:

```
try:
    print(gethostbyname('Ken'))
except Exception as exception:
    print(exception)
```

When developing a network application, the programmer can first try it out on a **local host**—that is, on a standalone computer that may or may not be connected to the Internet. The computer's IP name in this case is '**localhost**'. The IP number of a computer that acts as a local host is distinct from its IP number as an **Internet host**, as shown in the next session:

```
>>> gethostbyname(gethostname())
'192.168.1.109'
>>> gethostbyname('localhost')
'127.0.0.1'
>>>
```

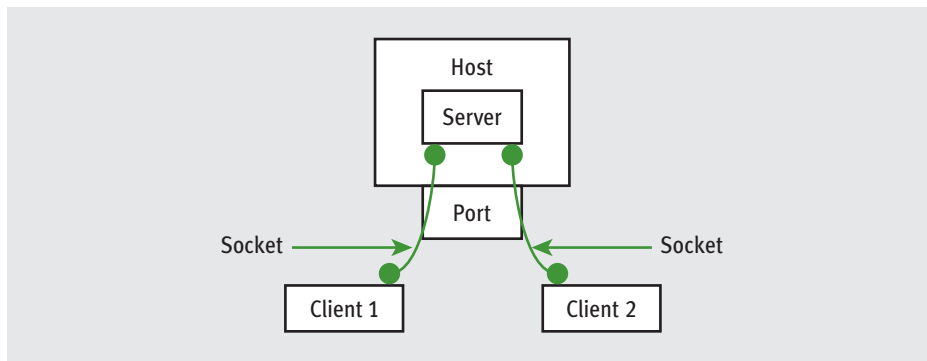
When the programmer is satisfied that the application is working correctly on a local host, the application can then be deployed on the Internet host simply by changing the IP address. In the discussion that follows, we use a local host to develop network applications.

10.2.2 Ports, Servers, and Clients

Clients connect to servers via objects known as **ports**. A port serves as a channel through which several clients can exchange data with the same server or with different servers. Ports are usually specified by numbers. Some ports are dedicated to special servers or tasks. For example, almost every computer reserves port number 13 for the day/time server, which allows clients to obtain the date and time. Port number 80 is reserved for a Web server, and so forth. Most computers also have hundreds or even thousands of free ports available for use by any network applications.

10.2.3 Sockets and a Day/Time Client Script

You can write a Python script that is a client to a server. To do this, you need to use a **socket**. A socket is an object that serves as a communication link between a single server process and a single client process. You can create and open several sockets on the same port of a host computer. Figure 10.4 shows the relationships between a host computer, ports, servers, clients, and sockets.



[FIGURE 10.4] Setup of day/time host and clients

A Python day/time client script uses the **socket** module introduced earlier. This script does the following:

- Creates a socket object.
- Opens the socket on a free port of the local host. We use a large number, 5000, for this port.
- Reads and decodes the day/time from the socket.
- Displays the day/time.

Here is a Python script that performs these tasks:

```
"""
Client for obtaining the day and time.
"""

from socket import *
from codecs import decode

HOST = 'localhost'
```

continued

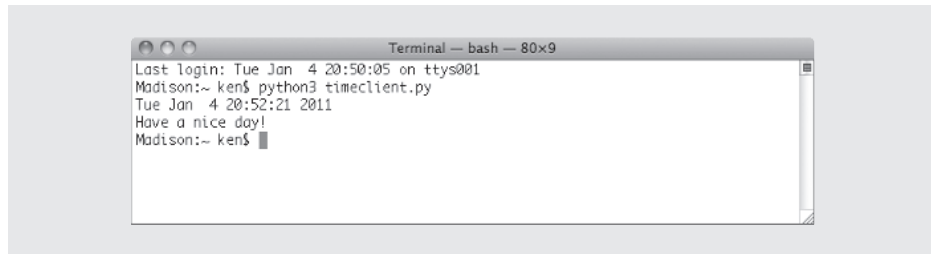
```

PORT = 5000
BUFSIZE = 1024
ADDRESS = (HOST, PORT)

server = socket(AF_INET, SOCK_STREAM)           # Create a socket
server.connect(ADDRESS)                         # Connect it to a host
dayAndTime = decode(server.recv(BUFSIZE), 'ascii') # Read and decode a string
print(dayAndTime)
server.close()                                  # Close the connection

```

Although we cannot run this script until we write and launch the server program, Figure 10.5 shows the client's anticipated output.



[FIGURE 10.5] The interface of the day/time client script

As you can see, a Python socket is fairly easy to set up and use. A socket resembles a file object, in that the programmer opens it, receives data from it, and closes it when finished. We now explain these steps in our client script in more detail.

The script creates a socket by running the function **socket** in the **socket** module. This function returns a new socket object, when given a socket family and a socket type as arguments. We use the family **AF_INET** and the type **SOCK_STREAM**, both **socket** module constants, in all of our examples.

To connect the socket to a host computer, one runs the socket's **connect** method. This method expects as an argument a tuple containing the host's IP address and a port number. In this case, these items are **'localhost'** and **5000**, respectively. These two items should be the same as the ones used in the server script.

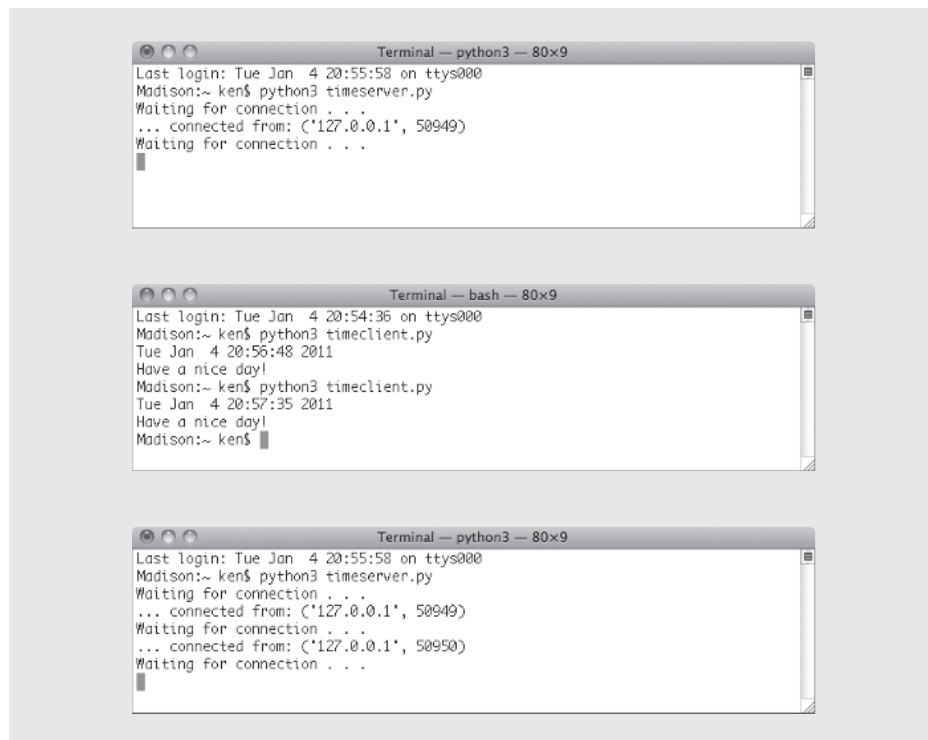
To obtain information sent by the server, the client script runs the socket's **recv** method. This method expects as an argument the maximum size in bytes of the data to be read from the socket. The **recv** method returns a **bytes** object. This object is converted to a string by calling the **codecs** function **decode**, with the encoding **'ascii'** as the second argument.

After the client script has printed the string read from the socket, the script closes the connection to the server by running the socket's `close` method.

10.2.4

A Day/Time Server Script

You can also write a day/time server script in Python to handle requests from many clients. Figure 10.6 shows the interaction between a day/time server and two clients in a series of screenshots. In the first shot, the day/time server script is launched in a terminal window, and it's waiting for a connection. In the second shot, two successive clients are launched in a separate terminal window (you can open several terminal windows at once). They have connected to the server and have received the day/time. The third shot shows the updates to the server's window after it has served these two clients. Note that the two clients terminate execution after they print their results, whereas the server appears to continue waiting for another client.



[FIGURE 10.6] A day/time server and two clients

A Python day/time server script also uses the resources of the **socket** module. The basic sequence of operations for a simple day/time server script is the following:

Create a socket and open it on port 5000 of the local host

While true:

 Wait for a connection from a client

 When the connection is made, send the date to the client

Our script also displays information about the host, the port, and the client. Here is the code, followed by a brief explanation:

```
"""
Server for providing the day and time.
"""

from socket import *
from time import ctime

HOST = 'localhost'
PORT = 5000
ADDRESS = (HOST, PORT)

server = socket(AF_INET, SOCK_STREAM)
server.bind(ADDRESS)
server.listen(5)

while True:
    print('Waiting for connection . . .')
    client, address = server.accept()
    print('... connected from:', address)
    client.send(bytes(ctime() + '\nHave a nice day!', 'ascii'))
    client.close()
```

The server script uses the same information to create a socket object as the client script presented earlier. In particular, the IP address and port number must be *exactly* the same as they are in the client's code.

However, connecting the socket to the host and to the port so as to become a server socket is done differently. First, the socket is bound to this address by running its **bind** method. Second, the socket then is made to listen for up to five requests from clients by running its **listen** method.

After the script enters its main loop, it prints a message indicating that it is waiting for a connection. The socket's **accept** method then pauses execution of the script, in a manner similar to Python's **input** function, to wait for a request from a client.

When a client connects to this server, **accept** returns a tuple containing the client's socket and its address information. Our script binds the variables **client** and **address** to these values and uses them in the next steps.

The script prints the client's address, and then sends the current day/time to the client by running the **send** method with the client's socket. The **send** method expects a **bytes** object as an argument. A **bytes** object is created from a string by calling the built-in **bytes** function, with the string and an encoding, in this case, **'ascii'**, as arguments. The Python function **time.ctime** returns a string representing the day/time.

Finally, the script closes the connection to the client by running the client socket's **close** method. The script then returns in its infinite loop to accept another client connection.

10.2.5 A Two-Way Chat Script

The communication between the day/time server and its client is one-way. The client simply receives a message from the server and then quits. In a two-way chat, the client connects to the server, and the two programs engage in a continuous communication until one of them, usually the client, chooses to quit.

Once again, there are two distinct Python scripts, one for the server and one for the client. The setup of a two-way chat server is similar to that of the day/time server discussed earlier. The server script creates a socket with a given IP address and port and then enters an infinite loop to accept and handle clients. When a client connects to the server, the server sends the client a greeting.

Instead of closing the client's socket and listening for another client connection, the server then enters a second, nested loop. This loop engages the server in a continuous conversation with the client. The server receives a message from the client. If the message is an empty string, the server displays a message that the client has disconnected, closes the client's socket, and breaks out of the nested loop. Otherwise, the server prints the client's message and prompts the user for a reply to send to the client.

Here is the code for the two loops in the server script:

```
while True:
    print('Waiting for connection . . .')
    client, address = server.accept()
    print('... connected from:', address)
    client.send(bytes('Welcome to my chat room!', 'ascii')) # Send greeting

    while True:
        message = decode(client.recv(BUFSIZE), 'ascii') # Reply from client
        if not message:
            print('Client disconnected')
            client.close()
            break
        else:
            print(message)
            client.send(bytes(input('> '), 'ascii')) # Reply to client
```

The client script for the two-way chat sets up a socket in a similar manner to the day/time client. After the client has connected to the server, it receives and displays the server's initial greeting message.

Instead of closing the server's socket, the client then enters a loop to engage in a continuous conversation with the server. This loop mirrors the loop that is running in the server script. The client's loop prompts the user for a message to send to the server. If this string is empty, the loop breaks. Otherwise, the client sends the message to the server's socket and receives the server's reply. If this reply is the empty string, the loop also breaks. Otherwise, the server's reply is displayed. The server's socket is closed after the loop has terminated. Here is the code for the part of the client script following the client's connection to the server:

```
print(decode(server.recv(BUFSIZE), 'ascii')) # The server's greeting
while True:
    message = input('> ') # Get my reply or quit
    if not message:
        break
    server.send(bytes(message, 'ascii')) # Send my reply to the server
    reply = decode(server.recv(BUFSIZE), 'ascii') # Get the server's reply
    if not reply:
        print('Server disconnected')
        break
    print(reply) # Display the server's reply
server.close()
```

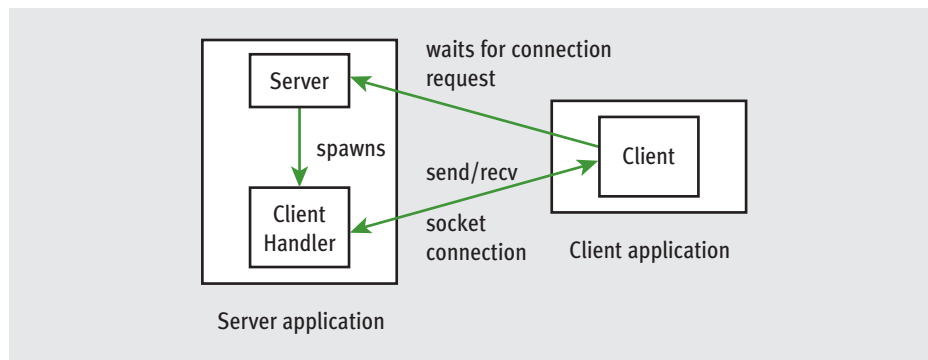

As you can see, it is important to synchronize the sending and the receiving of messages between the client and the server. If you get this right, the conversation can proceed, usually without a hitch.

10.2.6

Handling Multiple Clients Concurrently

The client/server programs that we have discussed thus far are rather simple and limited. First, the server handles a client's request and then returns to wait for another client. In the case of the day/time server, the processing of each request happens so quickly that clients will never notice a delay. However, when a server provides extensive processing, other clients will have to wait until the currently connected client is finished.

To solve the problem of giving many clients timely access to the server, we relieve the server of the task of handling the client's request and assign it instead to a separate client-handler thread. Thus, the server simply listens for client connections and dispatches these to new client-handler objects. The structure of this system is shown in Figure 10.7.



[FIGURE 10.7] A day/time server with a client handler

For our first example, let's modify the day/time server script by adding a client handler. This handler is an instance of a new class, **ClientHandler**, defined in the server's script. This class extends the **Thread** class. Its constructor receives the client's socket from the server and assigns it to an instance variable.

The **run** method includes the code to send the date to the client and close its socket. Here is the code for the complete, revised day/time server script:

```
"""
Server for providing the day and time.  Uses client
handlers to handle clients' requests.
"""

from socket import *
from time import ctime
from threading import Thread

class ClientHandler(Thread):
    """Handles a client request."""
    def __init__(self, client):
        Thread.__init__(self)
        self._client = client

    def run(self):
        self._client.send(bytes(ctime() + '\nHave a nice day!', 'ascii'))
        self._client.close()

HOST = 'localhost'
PORT = 5000
ADDRESS = (HOST, PORT)

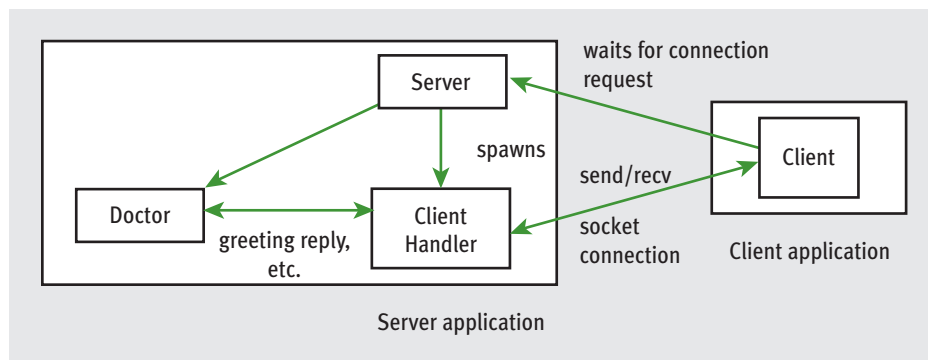
server = socket(AF_INET, SOCK_STREAM)
server.bind(ADDRESS)
server.listen(5)

# The server now just waits for connections from clients
# and hands sockets off to client handlers
while True:
    print('Waiting for connection . . .')
    client, address = server.accept()
    print('... connected from:', address)
    handler = ClientHandler(client)
    handler.start()
```

The code for the client's script does not change at all.

10.2.7 Setting Up Conversations for Others

Now that we have modified the day/time server to handle multiple clients, can we also modify the two-way chat program to support chats among multiple clients? Let us consider first the problem of supporting multiple two-way chats. We don't want to involve the server in the chat, much less the human user who is running the server. Can we first set up a chat between a human user and an automated agent? The doctor program developed in Case Study 5.5 in Chapter 5 is a good example of an automated agent that chats with its client, who is a human user. Building on this interaction, a doctor server program listens for requests from clients for doctors. Upon receiving a request, the server dispatches the client's socket and a new **Doctor** object (see Programming Project 9 in Chapter 8) to a handler thread. This thread then manages the conversation between this doctor and the client. The server returns to field more requests from clients for sessions with doctors. Figure 10.8 shows the structure of this program.



[FIGURE 10.8] The structure of a client/server program for patients and doctors

In the code that follows, we assume that a **Doctor** class is defined in the module **doctor.py**. This class includes two methods. The method **greeting** returns a string representing the doctor's welcome. The method **reply** expects the patient's string as an argument and returns the doctor's response string. The patient or client signals the end of a session by simply pressing the return key, which causes the client script's loop to terminate and close its connection to the server. Thus, the client script for this program is exactly the same as the client script for the two-way chat program. The server script combines elements of the

two-way chat server and the day/time server for multiple clients. The client handler resembles the one in the day/time server, but includes the following changes:

- The client handler's `__init__` method receives a **Doctor** object from the server and assigns it to an extra instance variable.
- The client handler's `run` method includes a conversation management loop similar to the one in the chat server. However, when the client handler receives a message from the client socket, this message is sent to the **Doctor** object rather than displayed in the server's terminal window. Then, instead of taking input from the server's keyboard and sending it to the client, the server obtains this reply from the **Doctor** object.

Here is the code for the server, as defined in `doctorserver.py`:

```
"""
File: doctorserver.py

Server for a therapy session. Handles multiple clients
concurrently.
"""

from socket import *
from codecs import decode
from threading import Thread
from doctor import Doctor

class ClientHandler(Thread):
    """Handles a session between a doctor and a patient."""
    def __init__(self, client, dr):
        Thread.__init__(self)
        self._client = client
        self._dr = dr

    def run(self):
        self._client.send(bytes(self._dr.greeting(), 'ascii'))
        while True:
            message = decode(self._client.recv(BUFSIZE), 'ascii')
            if not message:
                print('Client disconnected')
                self._client.close()
                break
            else:
                self._client.send(bytes(self._dr.reply(message), 'ascii'))
```

continued

```

HOST = 'localhost'
PORT = 5000
ADDRESS = (HOST, PORT)
BUFSIZE = 1024

server = socket(AF_INET, SOCK_STREAM)
server.bind(ADDRESS)
server.listen(5)

while True:
    print('Waiting for connection . . .')
    client, address = server.accept()
    print('... connected from:', address)
    dr = Doctor()
    handler = ClientHandler(client, dr)
    handler.start()

```

10.2 Exercises

- 1 Explain the role that ports and IP addresses play in a client/server program.
- 2 What is a local host, and how is it used to develop networked applications?
- 3 Why is it a good idea for a server to create threads to handle clients' requests?
- 4 Describe how a menu-driven command processor of the type developed for an ATM application in Chapter 8 could be run on a network.
- 5 The servers discussed in this section all contain infinite loops. Thus, the applications running them cannot do anything else while the server is waiting for a client's request, and they cannot even gracefully be shut down. Suggest a way to restructure these applications so that the applications can do other things, including performing a graceful shutdown.

10.3

Case Study: A Multi-Client Chat Room

Chat servers can also support chats among multiple clients. In this case study, we develop a client/server application that supports a chat room for two or more participants.

10.3.1

Request

Write a program that supports an online chat room.

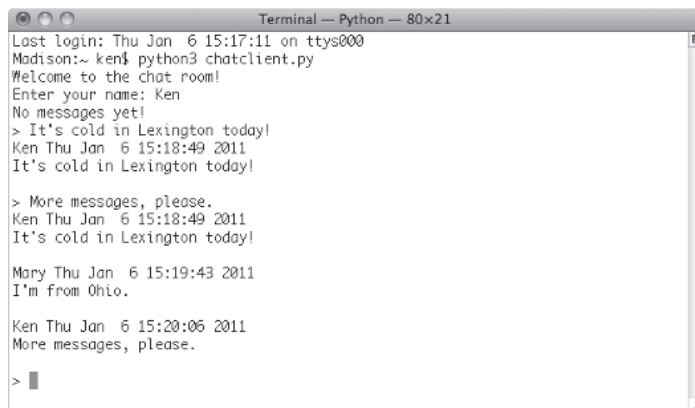
10.3.2

Analysis

The server is started like the other servers discussed in this chapter. When a client connects, it prompts its human user for a user name and sends this string to the server. The client then receives a welcome from the server and a message containing a record of the conversation thus far. This record includes zero or more chunks of text, each of which has the following format:

```
<day/time> <user name>  
<message>
```

The client can then join the conversation by sending a message to the server. The server receives this message, adds it to the common record, and sends that record back to the client. Thus, a client receives an updated record whenever it sends a message to the server. Furthermore, this record contains the messages of any number of clients that have joined in the conversation since the server started. A session for a client is shown in Figure 10.9.

A terminal window titled "Terminal — Python — 80x21" showing a chat client session. The session starts with a login message: "Last login: Thu Jan 6 15:17:11 on ttys000". The user runs "python3 chatclient.py", which prompts "Welcome to the chat room!" and "Enter your name: Ken". The user then enters "It's cold in Lexington today!". The terminal shows the message being received by the client: "Ken Thu Jan 6 15:18:49 2011" and "It's cold in Lexington today!". The user then enters "More messages, please.", and the terminal shows the message being received by the client: "Ken Thu Jan 6 15:18:49 2011" and "It's cold in Lexington today!". The user then enters "Mary Thu Jan 6 15:19:43 2011" and "I'm from Ohio.". The terminal shows the message being received by the client: "Ken Thu Jan 6 15:20:06 2011" and "More messages, please.". The user then enters ">".

```
Terminal — Python — 80x21
Last login: Thu Jan 6 15:17:11 on ttys000
Madison:~ ken$ python3 chatclient.py
Welcome to the chat room!
Enter your name: Ken
No messages yet!
> It's cold in Lexington today!
Ken Thu Jan 6 15:18:49 2011
It's cold in Lexington today!

> More messages, please.
Ken Thu Jan 6 15:18:49 2011
It's cold in Lexington today!

Mary Thu Jan 6 15:19:43 2011
I'm from Ohio.

Ken Thu Jan 6 15:20:06 2011
More messages, please.

> █
```

[FIGURE 10.9] A client's session with the multi-client chat room program

The classes for this program are named **ClientHandler** and **ChatRecord**. They have roles similar to those of the **ClientHandler** and the **Doctor** classes in an earlier example, but there is just a single shared instance of **ChatRecord** for all clients.

10.3.3 Design

This chat room program's structure and behavior are similar to those of the online therapy server described earlier in this chapter. However, instead of communicating with a single autonomous software agent (a doctor), a client communicates with the other clients. They do so by sharing a common record or transcript of their conversation. At program startup, the server creates an instance of the **ChatRecord** class and assigns this object to a module variable. The server then passes this single record to the client handler for each new client that connects to the server.

The client handler maintains instance variables for the **ChatRecord** and its client's user name. When the handler receives a message from its client, this message is stamped with the user name and current day/time. The resulting chunk of text is then added to the common record. The text of the entire record is then sent back to the client.

The design of the program lends it nicely to the addition of other features, such as saving the chat record in a text file. Other improvements include splitting

the client's inputs and the server's outputs into separate text areas with a GUI, as described in Chapter 9.

10.3.4 Implementation (Coding)

We present first the code for the client script. This script differs a bit from our earlier examples, because it must prompt the human user for a user name and send it to the server before entering its conversation loop. Otherwise, there are no important changes.

```
"""
Client for a multi-client chat room.
"""

from socket import *
from codecs import decode

HOST = 'localhost'
PORT = 5000
BUFSIZE = 1024
ADDRESS = (HOST, PORT)
CODE = 'ascii'
server = socket(AF_INET, SOCK_STREAM)
server.connect(ADDRESS)
print(decode(server.recv(BUFSIZE), CODE))
name = input('Enter your name: ')
server.send(bytes(name, CODE))

while True:
    record = decode(server.recv(BUFSIZE), CODE)
    if not record:
        print('Server disconnected')
        break
    print(record)
    message = input('> ')
    if not message:
        print('Server disconnected')
        break
    server.send(bytes(message, CODE))
server.close()
```

The server script includes a definition of the **ClientHandler** class, which manages the conversation for a particular client.


```

"""
Server for a multi-client chat room.
"""

from socket import *
from codecs import decode
from chatrecord import ChatRecord
from threading import Thread
from time import ctime

class ClientHandler(Thread):

    def __init__(self, client, record):
        Thread.__init__(self)
        self._client = client
        self._record = record

    def run(self):
        self._client.send(bytes('Welcome to the chat room!', CODE))
        self._name = decode(self._client.recv(BUFSIZE), CODE)
        self._client.send(bytes(str(self._record), CODE))
        while True:
            message = decode(self._client.recv(BUFSIZE), CODE)
            if not message:
                print('Client disconnected')
                self._client.close()
                break
            else:
                message = self._name + ' ' + \
                    ctime() + '\n' + message
                self._record.add(message)
                self._client.send(bytes(str(self._record), CODE))

HOST = 'localhost'
PORT = 5000
ADDRESS = (HOST, PORT)
BUFSIZE = 1024
CODE = 'ascii'
record = ChatRecord()
server = socket(AF_INET, SOCK_STREAM)
server.bind(ADDRESS)
server.listen(5)

while True:
    print('Waiting for connection ...')
    client, address = server.accept()
    print('... connected from:', address)
    handler = ClientHandler(client, record)
    handler.start()

```

The **ChatRecord** class is defined in the file **chatrecord.py**. The class is rather simple, but can be refined to manage other potential extensions to the program, such as searches for a given user's messages. Here is the code:

```
class ChatRecord(object):

    def __init__(self):
        self.data = []

    def add(self, s):
        self.data.append(s)

    def __str__(self):
        if len(self.data) == 0:
            return 'No messages yet!'
        else:
            return '\n'.join(self.data)
```

You might have noticed that the chat record is actually shared among several client-handler threads. This presents a potential synchronization problem of the type discussed earlier in this chapter. If one handler is timed-out in the middle of a mutation to the record, some data might be lost or corrupted for this or other clients. The solution of this problem is left as an exercise for you.

Summary

- Threads allow the work of a single program to be distributed among several computational processes. These processes may be run concurrently on the same computer or may collaborate by running on separate computers.
- A thread can have several states during its lifetime, such as born, ready, executing (in the CPU), sleeping, and waiting. The queue schedules the threads in first-come, first-served order.
- After a thread is started, it goes to the end of the ready queue to be scheduled for a turn in the CPU.
- A thread may give up the CPU when it is timed-out, goes to sleep, waits on a condition, or finishes its **run** method.
- When a thread wakes up, is timed-out, or is notified that it can stop waiting, it returns to the rear of the ready queue.

- Thread synchronization problems can occur when two or more threads share data. These threads can be synchronized by waiting on conditions that control access to the data.
- Each computer on a network has a unique IP address that allows other computers to locate it. An IP address contains an IP number, but can also be labeled with an IP name.
- Servers and clients can communicate on a network by means of sockets. A socket is created with a port number and an IP address of the server on the client's computer and on the server's computer.
- Clients and servers communicate by sending and receiving bytes through their socket connections. A string is converted to bytes before being sent, and the bytes are converted to a string after receipt.
- A server can handle several clients concurrently by assigning each client request to a separate handler thread.

REVIEW QUESTIONS

- 1 Multiple threads can run on the same desktop computer by means of
 - a timesharing
 - b multiprocessing
 - c distributed computing
- 2 A **Thread** object moves to the ready queue when
 - a its **wait** method is called
 - b its **sleep** method is called
 - c its **start** method is called
- 3 The method that executes a thread's code is called
 - a the **start** method
 - b the **run** method
 - c the **execute** method
- 4 A lock on a resource is provided by an instance of the
 - a **Thread** class
 - b **Condition** class
 - c **Lock** class

- 5 If multiple threads share data, they can have
 - a total cooperation
 - b synchronization problems
- 6 The object that uniquely identifies a host computer on a network is a(n)
 - a port
 - b socket
 - c IP address
- 7 The object that allows several clients to access a server on a host computer is a(n)
 - a port
 - b socket
 - c IP address
- 8 The object that effects a connection between an individual client and a server is a(n)
 - a port
 - b socket
 - c IP address
- 9 The data that are transmitted between client and server are
 - a of any type
 - b strings
- 10 The best way for a server to handle requests from multiple clients is to
 - a directly handle each client's request
 - b create a separate client-handler thread for each client

PROJECTS

- 1 Redo the producer/consumer program so that it allows multiple consumers. Each consumer must be able to consume the same data before the producer produces more data.
- 2 Assume that there are five sections of Computer Science 101, each with 20 spots for students. The computer application that assigns students to course sections includes requests from multiple threads for spots in the course. Write a program that allows 100 concurrently running student threads to request and obtain spots, in such a manner that the enrollment of no course exceeds the limit.
- 3 Restructure one of the network applications discussed in this chapter so that it can be shut down gracefully.
- 4 The game of craps, which was developed as a program in Chapter 8, can involve two players. Restructure that program as a network application, so that a client can play against the server. The client gets to roll first, and then it and the server alternate. The first player to get a winning roll wins, whereas the first player to get a losing roll loses. The two players each have their own set of dice. (*Hint:* The client handler on the server side maintains the two **Player** objects for the game, and each **Player** object should perform one roll at a time. The client signals a new roll by pressing the enter key, whereas the server rolls automatically.)
- 5 Modify the multi-client chat room application discussed in this chapter so that it maintains the chat record in a text file. The record should load the text from the file at instantiation and save each message as it is received.
- 6 In the multi-client chat room application, a client must send a message to the server to receive a record of the chat. Suggest and implement a way for the client to receive the chat record even if it has nothing significant to say.
- 7 Modify the network application for therapy discussed in this chapter so that it handles multiple clients. Each client has its own doctor object. The program saves the doctor object for a client when it disconnects. Each doctor object should be associated with a patient user name. When a new patient logs in, a new doctor is created. But when an existing patient logs in, its doctor object is read from a file having that patient's user name. Each doctor object should have its own history list of a patient's inputs for generating replies that refer to earlier conversations.

- 8 Design, implement, and test a network application that maintains an online phonebook. The data model for the phonebook is saved in a file on the server's computer. Clients should be able to look up a person's phone number or add a name and number to the phonebook. The server should handle multiple clients without delays.
- 9 Convert the ATM application presented in Chapter 8 to a networked application. The client manages the user interface, whereas the server handles transactions with the bank.
- 10 Modify the programs of Project 8.5 and Project 10.9 so that the ATM server is one component of a larger application that manages the bank. The bank manager should allow the user to view, modify, add, and remove accounts as well as launch or shut down the ATM server.

[APPENDIX]

A

Python Resources

Table A.1 provides information on an excellent Web site where programmers can find complete documentation for the Python API (Application Programming Interface) and download Python and other resources.

DESCRIPTION	URL	EXPLANATION
Python's top-level Web page	http://www.python.org/	This page contains news about events in the Python world and links to documentation, Python-related products, program examples, and free downloads of resources.
Downloads	http://www.python.org/download/	This page allows you to select the version of Python that matches your computer and to begin the download process.
Documentation and training	http://www.python.org/doc/	This page allows you to browse the documentation for the Python API, tutorials, and other training aids. You can also download many of these items to your computer for offline reference.

[TABLE A.1] Online Python Documentation

The following sections discuss some situations that involve downloading files or information from the Web.

A.1

Installing Python on Your Computer

As of this writing, Python does not come preinstalled on Windows systems. Therefore, you must download the Windows installer from <http://www.python.org/download/>. The installer might then run automatically, or you might have to double-click an icon for the installer to launch it. The installer automatically puts Python into a folder and inserts various command options on the **All Programs** menu. Note that administrators installing Python for all users on Windows Vista need to be logged in as Administrator.

Macintosh users running Mac OS X might need to update the version of Python that comes preinstalled on their systems. A Mac OS X installer can be downloaded for this purpose and behaves in a manner similar to that of the Windows installer.

Unix and Linux users also might need to upgrade the version of Python that comes preinstalled on their systems. In these cases, they have to download a compressed Python source code “tarball” from the same site and install it.

Most users will also want to place aliases of the important Python commands, such as the one that launches IDLE, on their desktops.

A.2

Using the Terminal Command Prompt, IDLE, and Other IDEs

To launch an interactive session with Python’s shell from a terminal command prompt, open a terminal window, and enter **python** or **python3** at the prompt. To end the session on Unix machines (including Mac OS X), press the Control+D key combination at the session prompt. To end a session on Windows, press Control+Z, and then press Enter.

Before you run a Python script from a terminal command prompt, the script file must be in the current working directory, or the system path must be set to the file’s directory. You should consult your system’s documentation on how to set a path. To run a script, enter **python** or **python3**, followed by a space, followed by the name of the script’s file (including the **.py** extension), followed by any command-line arguments that the script expects.

On Windows, you can also launch a Python script by double-clicking the script's file icon. On Macintosh, Unix, and Linux systems, you must first configure the system to launch Python when files of this type are launched. The **File/Get Info** option on a Macintosh, for example, allows you to do this. You can also configure your system to launch Python using the simpler **python** command rather than **python3**.

You can also launch an interactive session with a Python shell by launching IDLE (as of this writing, the specific command to run is **idle3.1**). There are many advantages to using an IDLE shell rather than a terminal-based shell, such as color-coded program elements, menu options for editing code and consulting documentation, and the ability to repeat commands.

IDLE also helps you manage program development with multiple editor windows. You can run code from these windows and easily move code among them. Although this book does not discuss it, a debugging tool is also available within IDLE.

There are several other free and commercial IDEs with capabilities that extend those of IDLE. jEdit (<http://www.jedit.org/>) is a free, lightweight IDE that has widespread use in academic environments because it also supports Java and C++ program development.

[APPENDIX]

B

INSTALLING THE **images** Library

The **images** library is a nonstandard, open-source Python module developed to support easy image processing.

The **images** library supports the processing of GIF images only. The source code for the library, in the file **images.py**, is available on the author's Web site at <http://home.wlu.edu/~lambertk/python/>, or from your instructor.

In general, there are two ways to install a Python library:

- 1 Place the source file for the library in the current working directory. Then, when you launch a Python script from this directory or load it from an IDLE window into a shell, Python can locate the library resources that are imported by that script. The disadvantage of this installation option is that the library must be moved whenever you change working directories.
- 2 Place the source file in the directory that Python has established for third-party libraries. The path to this directory will vary, depending on your system. For Windows users, this path will be something like **c:\python31\Lib\site-packages**. For Unix or Macintosh users, it might be something like **/usr/local/bin/lib/python3.1/site-packages**. Once a library is placed in this directory, a Python script can access its resources from any directory on your system.

The **images.py** file can be installed using one of the preceding methods, and your client code will be ready to use this module.

[APPENDIX]

C

THE API FOR Image Processing

Both the graphics and the image-processing library are based on Python's standard **tkinter** library. The API (Application Programming Interface) for the image-processing library follows.

The **images** module includes a single class named **Image**. Each **Image** object represents an image. The programmer can supply the filename of an image on disk when **Image** is instantiated. The resulting **Image** object contains pixels loaded from an image file on disk. If a filename is not specified, a height and width must be specified. The resulting **Image** object contains the specified number of pixels with a single default color.

When the programmer imports the **Image** class and instantiates it, no window opens. At that point, the programmer can run various methods with this **Image** object to access or modify its pixels, as well as save the image to a file. At any point in your code, you may run the **draw** method with an **Image** object. At this point, a window will open and display the image. The program then waits for you to close the window before allowing you, either in the shell or in a script, to continue running more code.

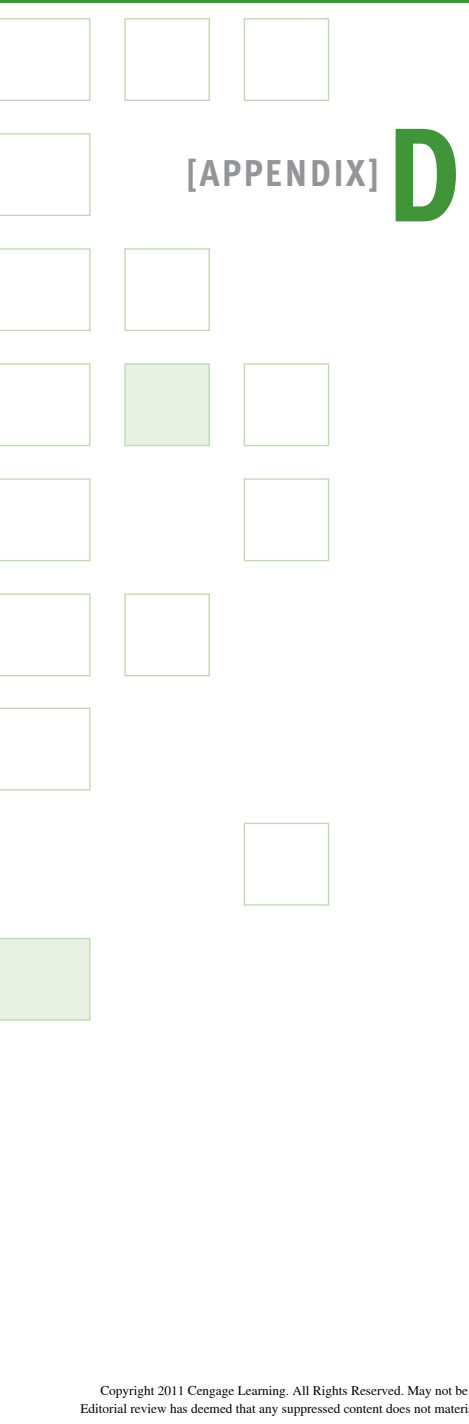
The positions of pixels in an image are the same as screen coordinates for display in a window. That is, the origin (0, 0) is in the upper-left corner of the image, and its (width, height) is in the lower-right corner.

Images can be manipulated either interactively within a Python shell or from a Python script. We recommend that the shell or script be launched from a system terminal, rather than from IDLE.

Image objects cannot be viewed in multiple windows at the same time from the same script. If you want to view two or more **Image** objects simultaneously, you can create separate scripts for them and launch these **Image** objects in separate terminal windows.

As mentioned earlier, the **images** module supports the use of GIF files only. Here is a list of the **Image** methods:

- **Image(filename)** Loads an image from the file named **filename** and returns an **Image** object that represents this image. The file must exist in the current working directory.
- **Image(width, height)** Returns an **Image** object of the specified width and height with a single default color.
- **getWidth()** Returns the width of the image in pixels.
- **getHeight()** Returns the height of the image in pixels.
- **getPixel(x, y)** Returns the pixel at the specified coordinates. A pixel is of the form (r, g, b), where the letters are integers representing the red, green, and blue values of a color in the RGB system.
- **setPixel(x, y, (r, g, b))** Resets the pixel at position (x, y) to the color value represented by (r, g, b). The coordinates must be in the range of the image's coordinates, and the RGB values must range from 0 through 255.
- **draw()** Opens a window and displays the image. The user must close the window to continue the program.
- **save()** Saves the image to its current file, if it has one. Otherwise, it does nothing.
- **save(filename)** Saves the image to the given file and makes it the current file. This is similar to the **Save As** option in most **File** menus.



[APPENDIX]

D

TRANSITION FROM PYTHON TO Java and C++

Although Python is an excellent teaching language and is gaining acceptance in industry, Java and the C/C++ family of languages remain the most widespread languages used in higher education and real-world settings. Thus, computer science students must become proficient in these languages, both to continue in their course work and to prepare for careers in the field.

Fortunately, the transition from Python to Java or C++ is not difficult. Although the syntactic structures of Python and these other languages are somewhat different, the languages support the same programming styles. For an overview of all the essential differences between Python, Java, and C++, see the author's Web site at <http://home.wlu.edu/~lambertk/python/>.

GLOSSARY

A

- abacus** An early computing device that allowed users to perform simple calculations by moving beads along wires.
- abstract** Simplified or partial, hiding detail.
- abstract class** A class that defines attributes and methods for subclasses, but is never instantiated.
- abstraction** A simplified view of a task or data structure that ignores complex detail.
- accessor** A method used to examine an attribute of an object without changing it.
- activation record** An area of computer memory that keeps track of a function or method call's parameters, local values, return value, and the caller's return address. *See also* **run-time stack**.
- algorithm** A finite sequence of instructions that, when applied to a problem, will solve it.
- alias** A situation in which two or more names in a program can refer to the same memory location. An alias can cause subtle side effects.
- analysis** The phase of the software life cycle in which the programmer describes what the program will do.
- Analytical Engine** A general-purpose computer designed in the nineteenth century by Charles Babbage, but never completed.
- anonymous function** A function without a name, constructed in Python using **lambda**.
- application software** Programs that allow human users to accomplish specialized tasks, such as word processing or database management. Also called applications.
- argument** A value or expression passed in a method call.
- arithmetic expression** A sequence of operands and operators that computes a value.
- arithmetic overflow** A situation that arises when the computer's memory cannot

represent the number resulting from an arithmetic operation.

artificial intelligence A field of computer science whose goal is to build machines that can perform tasks that require human intelligence.

ASCII character set The American Standard Code for Information Interchange ordering for a character set.

assembler A program that translates an assembly language program to machine code.

assembly language A computer language that allows the programmer to express operations and memory addresses with mnemonic symbols.

assignment operator The symbol =, which is used to give a value to a variable.

assignment statement A method of giving values to variables.

association A pair of items consisting of a key and a value.

attribute A property that a computational object models, such as the balance in a bank account.

augmented assignment An assignment operation that performs a designated operation, such as addition, before storing the result in a variable.

B

base case The condition in a recursive algorithm that is tested to halt the recursive process.

batch processing The scheduling of multiple programs so that they run in sequence on the same computer.

behavior The set of actions that a class of objects supports.

binary digit A digit, either 0 or 1, in the binary number system. Program instructions are stored in memory using a sequence of binary digits. *See also bit.*

bit A binary digit.

bitmap A data structure used to represent the values and positions of points on a computer screen or image.

bit-mapped display screen A type of display screen that supports the display of graphics and images.

block An area of program text, offset by indentation, that contains statements and data declarations.

block cipher An encryption method that replaces characters with other characters located in a two-dimensional grid of characters.

Boolean expression An expression whose value is either true or false. *See also compound Boolean expression and simple Boolean expression.*

bottom-up implementation A method of coding a program that starts with lower-level modules and a test driver module.

button object A window object that allows the user to select an action by clicking a mouse.

byte A sequence of bits used to encode a character in memory.

byte code The kind of object code generated by a Python compiler and interpreted by a Python virtual machine. Byte code is platform independent.

C

Caesar cipher An encryption method that replaces characters with other characters a given distance away in the character set.

call Any reference to a function or method by an executable statement. Also referred to as **invoke**.

call stack The trace of function or method calls that appears when Python raises an exception during program execution.

card reader A device that inputs information from punched cards into the memory of a computer.

c-curve A fractal shape that resembles the letter C.

central processing unit (CPU) A major hardware component that consists of the arithmetic/logic unit and the control unit. Also sometimes called a **processor**.

character set The list of characters available for data and program statements.

class A description of the attributes and behavior of a set of computational objects.

class diagram A graphical notation that describes the relationships among the classes in a software system.

class variable A variable that is visible to all instances of a class and is accessed by specifying the class name.

client A computational object that receives a service from another computational object.

client/server relationship A means of describing the organization of computing resources in which one resource provides a service to another resource.

coding The process of writing executable statements that are part of a program to solve a problem. *See also implementation.*

comments Nonexecutable statements used to make a program more readable.

compiler A computer program that automatically converts instructions in a high-level language to machine language.

compound Boolean expression Refers to the complete expression when logical connectives and negation are used to generate Boolean values. *See also Boolean expression and simple Boolean expression.*

computing agent The entity that executes instructions in an algorithm.

concatenation An operation in which the contents of one data structure are placed after the contents of another data structure.

concrete class A class that can be instantiated. *See also abstract class.*

concurrent processing The simultaneous performance of two or more tasks.

conditional statement *See* **selection statement**.

conjunction The connection of two Boolean expressions using the logical operator **and**, returning false if at least one of the expressions is false or true if they are both true.

constructor A method that is run when an object is instantiated, usually to initialize that object's instance variables. This method is named `__init__` in Python.

contained class A class that is used to define a data object within another class.

continuation condition A Boolean expression that is checked to determine whether or not to continue iterating within a loop. If this expression is true, iteration continues.

control statement A statement that allows the computer to repeat or select an action.

coordinate system A grid that allows a programmer to specify positions of points in a plane or of pixels on a computer screen.

correct program A program that produces an expected output for any legitimate input.

count-controlled loop A loop that stops when a counter variable reaches a specified limit.

CPU (central processing unit) A major hardware component that consists of the arithmetic/logic unit and the control unit. Also sometimes called a processor.

D

data The symbols that are used to represent information in a form suitable for storage, processing, and communication.

data decryption The process of translating encrypted data to a form that can be used.

data encryption The process of transforming data so that others cannot use it.

data structure A compound unit consisting of several data values.

data type(s) A set of values and operations on those values.

data validation The process of examining data prior to its use in a program.

debugging The process of eliminating errors, or “bugs,” from a program.

default behavior Behavior that is expected and provided under normal circumstances.

default parameter Also called a **default argument**. A special type of parameter that is automatically given a value if the caller does not supply one.

definite iteration The process of repeating a given action a preset number of times.

design The phase of the software life cycle in which the programmer describes how the program will accomplish its tasks.

design error An error such that a program runs, but unexpected results are produced. Also referred to as a logic error. *See also* **syntax error**.

dictionary A data structure that allows the programmer to access items by specifying key values.

docstring A sequence of characters enclosed in triple quotation marks (“”) that Python uses to document program components such as modules, classes, methods, and functions.

driver A method used to test other methods.

E

element A value that is stored in an array or a collection.

empty string A string that contains no characters.

encapsulation The process of hiding and restricting access to the implementation details of a data structure.

encryption The process of transforming data so that others cannot use it.

end-of-line comment Part of a single line of text in a program that is not executed, but serves as documentation for readers.

error See **design error** and **syntax error**.

escape sequence A sequence of two characters in a string, the first of which is `\`. The sequence stands for another character, such as the tab or newline.

event An occurrence, such as a button click or a mouse motion, that can be detected and processed by a program.

event-driven loop A process, usually hidden in the operating system, that waits for an event, notifies a program that an event has occurred, and returns to wait for more events.

exception An abnormal state or error that occurs during run time and is signaled by the operating system.

execute To carry out the instructions of a program.

expression A description of a computation that produces a value.

extended if statement Nested selection in which additional **if-else** statements are used in the **else** option. See also **nested if statement**.

external (or secondary) memory Memory that can store large quantities of data permanently.

F

Fibonacci numbers A series of numbers generated by taking the sum of the previous two numbers in the series. The series begins with the numbers 1 and 2.

field width The number of columns used for the output of text.

file A data structure that resides on a secondary storage medium.

file system Software that organizes data on secondary storage media.

filtering The successive application of a Boolean function to a sequence of arguments that returns a sequence of the arguments that make this function return **True**.

first-class data objects Data objects that can be passed as arguments to functions and returned as their values.

float A Python data type used to represent numbers with a decimal point, for example, a real number or a floating-point number.

floating-point number A data type that represents real numbers in a computer program.

for loop A structured loop consisting of an initializer expression, a termination expression, an update expression, and a statement.

format string A special syntax within a string that allows the programmer to specify the number of columns within which data are placed in a string.

fractal geometry A theory of shapes that are reflected in various phenomena, such as coastlines, water flow, and price fluctuations.

fractal object A type of mathematical object that maintains self-sameness when viewed at greater levels of detail.

front The end of a queue from which elements are removed.

function A chunk of code that can be treated as a unit and called to perform a task.

function heading The portion of a function implementation containing the function's name, parameter names, and return type.

G

garbage collection The automatic process of reclaiming memory when the data of a program no longer need it.

general method A method that solves a class of problems, not just one individual problem.

grammar The set of rules for constructing sentences in a language.

graphical user interface (GUI) See **GUI (graphical user interface)**

grid A data structure in which the items are accessed by specifying at least two index positions, one that refers to the item's row and another that refers to the item's column.

grid layout A Python layout class that allows the user to place window objects in a two-dimensional grid in the window.

GUI (graphical user interface) A means of communication between human beings and computers that uses a pointing device for input and a bitmapped screen for output. The bitmap displays images of windows and window objects such as buttons, text fields, and drop-down menus. The user interacts with the interface by using the mouse to directly manipulate the window objects. *See also* **window object**.

H

hacking The use of clever techniques to write a program, often for the purpose of gaining access to protected resources on networks.

hardware The computing machine and its support devices.

helper A method or function used within the implementation of a module or class but not used by clients of that module or class.

higher-order function A function that expects another function as an argument and/or returns another function as a value.

high-level programming language Any programming language that uses words and symbols to make it relatively easy to read and write a program. *See also* **assembly language** and **machine language**.

HTML (hypertext markup language) A programming language that allows the user to create pages for the World Wide Web.

hypermedia A data structure that allows the user to access different kinds of information (text, images, sound, video, applications) by traversing links.

hypertext A data structure that allows the user to access different chunks of text by traversing links.

hypertext markup language (HTML) *See* **HTML (hypertext markup language)**

I

identifiers Words that must be created according to a well-defined set of rules but can have any meaning subject to these rules.

identity The property of an object that it is the same thing at different points in time, even though the values of its attributes might change.

IDE (integrated development environment) A set of software tools that allows you to edit, compile, run, and debug programs within one user interface.

if-else statement A selection statement that allows a program to perform alternative actions based on a condition.

immutable object An object whose internal data or state cannot be changed.

implementation The phase of the software life cycle in which the program is coded in a programming language.

increment The process of increasing a number by 1.

indefinite iteration The process of repeating a given action until a condition stops the repetition.

index The relative position of a component of a linear data structure or collection.

indirect recursion A recursive process that results when one function calls another, which results at some point in a second call to the first function.

infinite loop A loop in which the controlling condition is not changed in such a manner to allow the loop to terminate.

infinite recursion In a running program, the state that occurs when a recursive method cannot reach a stopping state.

information hiding A condition in which the user of a module does not know the details of how it is implemented, and the implementer of a module does not know the details of how it is used.

information processing The transformation of one piece of information into another piece of information.

inheritance The process by which a subclass can reuse attributes and behavior defined in a superclass. *See also* **subclass** and **superclass**.

input Data obtained by a program during its execution.

input device A device that provides information to the computer. Typical input devices are a mouse, keyboard, disk drive, microphone, and network port. *See also* **I/O device** and **output device**.

instance A computational object bearing the attributes and behavior specified by a class.

instance variable Storage for data in an instance of a class.

instantiation The process of creating a new object or instance of a class.

integer A positive or negative whole number, or the number 0. The magnitude of an integer is limited by a computer's memory.

integer arithmetic operations Operations allowed on data of type `int`. These include the operations of addition, subtraction, multiplication, division, and modulus to produce integer answers.

integrated circuit The arrangement of computer hardware components in a single, miniaturized unit.

integrated development environment (IDE) *See* **IDE (integrated development environment)**

interface A formal statement of how communication occurs between the user of a module (class or method) and its implementer.

interpreter A program that translates and executes another program.

invoke *See* **call**.

I/O device Any device that allows information to be transmitted to or from a computer. *See also* **input device** and **output device**.

IP address The unique location of an individual computer on the Internet.

IP name A representation of an IP address that uses letters and periods.

IP number A representation of an IP address that uses digits and periods.

iteration *See* **loop**.

J

jump table A dictionary that associates command names with functions that are invoked when those functions are looked up in the table.

justification The process of aligning text to the left, the center, or the right within a given number of columns.

K

key An item that is associated with a value and which is used to locate that value in a collection.

keypunch machine An early input device that allowed the user to enter programs and data onto punched cards.

keywords *See* **reserved words**.

L

label object A window object that displays text, usually to describe the roles of other window objects.

lambda The mechanism by which an anonymous function is created.

left associative The property of an operator such that repeated applications of it are evaluated from left to right (first to last).

library A collection of methods and data organized to perform a set of related tasks. *See also* **class**.

lifetime The time during which a data object or method call exists.

linear An increase of work or memory in direct proportion to the size of a problem.

literal An element of a language that evaluates to itself, such as 34 or "hi there."

loader A system software tool that places program instructions and data into the appropriate memory locations before program start-up.

logical operator Any of the logical connective operators `&&` (and), `||` (or), or `!` (negation).

logical structure The organization of the components in a data structure, independent of their organization in computer memory.

loop A type of statement that repeatedly executes a set of statements.

loop body The action(s) performed on each iteration through a loop.

loop header Information at the beginning of a loop that includes the conditions for continuing the iteration process.

M

machine language The language used directly by the computer in all its calculations and processing. Also called **machine code**.

magnetic storage media Any media that allow data to be stored as patterns in a magnetic field.

main (primary or internal) memory The high-speed internal memory of a computer, also referred to as random access memory (RAM). *See also* **memory** and **secondary memory**.

main module The software component that contains the point of entry or start-up code of a program.

mainframe Large computers typically used by major companies and universities. *See also* **microcomputer** and **minicomputer**.

mapping The successive application of a function to a sequence of arguments that returns a sequence of results.

megabyte Shorthand for approximately 1 million bytes.

memory The ordered sequence of storage cells that can be accessed by address. Instructions and variables of an executing program are temporarily held here. *See also* **main memory** and **secondary memory**.

memory location A storage cell that can be accessed by address. *See also* **memory**.

method A chunk of code that can be treated as a unit and invoked by name. A method is called with an object or class.

method heading The portion of a method implementation containing the method's name, parameter declarations, and return type.

microcomputer A computer capable of fitting on a laptop or desktop, generally used by one person at a time. *See also* **mainframe** and **minicomputer**.

minicomputer A small version of a mainframe computer. It is usually used by several people at once. *See also* **mainframe** and **microcomputer**.

mixed-mode arithmetic Expressions containing data of different types; the values of these expressions will be of either type, depending on the rules for evaluating them.

model/view/controller pattern (MVC) A design plan in which the roles and responsibilities of the system are cleanly divided among data management (model), user interface display (view), and user event-handling (controller) tasks.

module An independent program component that can contain variables, functions, and classes.

Moore's Law A hypothesis that states that the processing speed and storage capacity of computers will increase by a factor of two every 18 months.

mutator A method used to change the value of an attribute of an object.

N

namespace The set of all of a program's variables and their values.

natural ordering The placement of data items relative to each other by some internal criteria, such as numeric value or alphabetical value.

negation The use of the logical operator **not** with a Boolean expression, returning **True** if the expression is false, and **False** if the expression is true.

nested if statement A selection statement used within another selection statement. *See also* **extended if statement**.

nested loop A loop as one of the statements in the body of another loop.

network A collection of resources that are linked together for communication.

newline character A special character ('\\n') used to indicate the end of a line of characters in a string or a file stream.

None value A special value that indicates that no object can be accessed.

O

object A collection of data and operations, in which the data can be accessed and modified only by means of the operations.

object code The code produced by a compiler.

object identity The property of an object that it is the same thing at different points in time, even though the values of its attributes might change.

object-based programming The construction of software systems that use objects.

object-oriented programming The construction of software systems that define classes and rely on inheritance and polymorphism.

off-by-one error Usually seen with loops, this error shows up as a result that is one less or one greater than the expected value.

operating system A large program that allows the user to communicate with the hardware and performs various management tasks.

optical storage media Devices such as CDs and DVDs that store data permanently and from which the data are accessed by using laser technology.

optional arguments Arguments to a function or method that may be omitted.

origin The point (0,0) in a coordinate system.

output Information that is produced by a program.

output device A device that allows you to see the results of a program. Typically, it is a monitor, printer, speaker, or network port. *See also* **input device** and **I/O device**.

overloading The process of using the same operator symbol or identifier to refer to many different functions. *See also* **polymorphism**.

overriding The process of re-implementing a method already implemented in a superclass.

P

parameter *See* **argument**.

parent The immediate superclass of a class.

path A sequence of edges that allows one vertex to be reached from another.

pixel A picture element or dot of color used to display images on a computer screen.

polymorphism The property of one operator symbol or method identifier having many meanings. *See also* **overloading**.

pop The operation that removes an element from a Python list or dictionary.

port A channel through which several clients can exchange data with the same server or with different servers.

positional notation The type of representation used in based number systems, in which the position of each digit denotes a power in the system's base.

precedence rules Rules that govern the order in which operators are applied in expressions.

predicate A function that returns a Boolean value.

prefix form The form of an expression in which the operator precedes its operands.

primary memory *See* **main memory** and **memory**.

problem decomposition The process of breaking a problem into subproblems.

problem instance An individual problem that belongs to a class of problems.

procedural programming A style of programming that decomposes a program into a set of methods or procedures.

program A set of instructions that tells the machine (the hardware) what to do.

program library Program code developed for use in other programs.

programming language Formal language that computer scientists use to give instructions to the computer.

prototype A trimmed-down version of a class or software system that still functions and allows the programmer to study its essential features.

pseudocode A stylized half-English, half-code language written in English but suggesting program code.

PVM (Python virtual machine) A program that interprets Python byte codes and executes them.

R

random access A data-access method that runs in constant time.

ready queue A data structure used to schedule processes or threads for CPU access.

rear The end of a queue to which elements are added.

recursion The process of a subprogram calling itself. A clearly defined stopping state must exist. Any recursive subprogram can be rewritten using **iteration**.

recursive call The call of a function that already has a call waiting in the current chain of function calls.

recursive definition A set of statements in which at least one statement is defined in terms of itself.

recursive design The process of decomposing a problem into subproblems of exactly the same form that can be solved by the same algorithm.

recursive function A function that calls itself.

recursive step A step in the recursive process that solves a similar problem of smaller size and eventually leads to a termination of the process.

recursive subprogram *See recursion.*

reducing The application of a function to a sequence of its arguments to produce a single value.

relational operator An operator used for comparison of data items of the same type.

repetition *See loops.*

required arguments Arguments that must be supplied by the programmer when a function or method is called.

reserved words Words that have predefined meanings that cannot be changed.

responsibility-driven design The assignment of roles and responsibilities to different actors in a program.

returning a value The process whereby a function or method makes the value that it computes available to the rest of the program.

root directory The top-level directory in a file system.

run-time stack An area of computer memory reserved for local variables and parameters of method calls.

run-time system Software that supports the execution of a program.

S

scientific notation The representation of a floating-point number that uses a decimal point and an exponent to express its value.

scope The area of program text in which the value of a variable is visible.

screen coordinate system A coordinate system used by most programming languages in which the origin is in the upper-left corner of the screen, window, or panel, and the y values increase toward the bottom of the drawing area.

script A Python program that can be launched from a computer's operating system.

secondary (external) memory An auxiliary device for memory, usually a disk or magnetic tape. *See also main memory and memory.*

selection The process by which a method or a variable of an instance or a class is accessed.

selection statement A control statement that selects some particular logical path based on the value of an expression. Also referred to as a **conditional statement**.

semantic error A type of error that occurs when the computer cannot carry out the instruction specified.

semantics The rules for interpreting the meaning of a program in a language.

semiconductor storage media Devices, such as flash sticks, that use solid state circuitry to store data permanently.

sentinel value (or **sentinel**) A special value that indicates the end of a set of data or of a process.

sequence A type of collection in which each item but the first has a unique predecessor and each item but the last has a unique successor.

server A computational object that provides a service to another computational object.

shell A program that allows users to enter and run Python program expressions and statements interactively.

short-circuit evaluation The process by which a compound Boolean expression halts evaluation and returns the value of the first subexpression that evaluates to true, in the case of **or**, or false, in the case of **and**.

side effect A change in a variable that is the result of some action taken in a program, usually from within a method.

simple Boolean expression An expression in which two numbers or variable values are compared using a single relational operator. *See also* **Boolean expression** and **compound Boolean expression**.

slicing An operation that returns a subsection of a linear collection, for example, a sublist or a substring.

socket An object that serves as a communication link between a single server process and a single client process.

software Programs that make the machine (the hardware) do something, such as word processing, database management, or games.

software development life cycle (SDLC) The process of development, maintenance, and demise of a software system. Phases include analysis, design, coding, testing/verification, maintenance, and obsolescence.

software reuse The process of building and maintaining software systems out of existing software components.

solid-state device An electronic device, typically based on a transistor, and which has no moving parts.

source code The program text as viewed by the human being who creates or reads it, prior to compilation.

source program A program written by a programmer.

stack frame *See* **activation record**.

stack overflow error A situation that occurs when the computer runs out of memory to allocate for its call stack. This situation usually arises during an infinite recursion.

state The set of all the values of the variables of a program at any point during its execution.

statement An individual instruction in a program.

step value The amount by which a counter is incremented or decremented in a count-controlled loop.

stepwise refinement The process of repeatedly subdividing tasks into subtasks until each subtask is easily accomplished. *See also* **structured programming** and **top-down design**.

string (string literal) One or more characters, enclosed in double quotation marks, used as a constant in a program.

strongly typed programming language A language in which the types of operands are checked prior to applying an operator to them, and which disallows such applications, either at run time or at compile time, when operands are not of the appropriate type.

structural equivalence A criterion of equality between two distinct objects in which one or more of their attributes are equal.

structure chart A graphical method of indicating the relationship between modules when designing the solution to a problem.

structured programming Programming that parallels a solution to a problem achieved by top-down implementation. *See also* **stepwise refinement** and **top-down design**.

subclass A class that inherits attributes and behaviors from another class.

subscript *See* **index**.

substring A string that represents a segment of another string.

superclass The class from which a subclass inherits attributes and behavior. *See also* **inheritance** and **subclass**.

symbolic constant A name that receives a value at program start-up and whose value cannot be changed.

synchronization problem A type of problem arising from the execution of threads or processes that share memory.

syntax The rules for constructing well-formed programs in a language. Also, the rules for forming sentences in a language.

syntax error An error in spelling, punctuation, or placement of certain key symbols in a program. *See also* **design error**.

system software The programs that allow users to write and execute other programs, including operating systems such as Windows and Mac OSX.

T

temporary variable A variable that is introduced in the body of a function or method for the use of that subroutine only.

terminal I/O interface A user interface that allows the user to enter input from a keyboard and view output as text in a window. Also called a **terminal-based interface**.

termination condition A Boolean expression that is checked to determine whether or not to stop iterating within a loop. If this expression is true, iteration stops.

test suite A set of test cases that exercise the capabilities of a software component.

text editor A program that allows the user to enter text, such as a program, and save it in a file.

text files Files that contain characters and are readable and writable by text editors.

text object A window object that provides a scrollable region within which the user can view or enter several lines of text.

thread A type of process that can run concurrently with other processes.

time sharing The scheduling of multiple programs so that they run concurrently on the same computer.

time-sharing operating system A computer system that can run multiple programs in such a manner that its users have the illusion that they are running simultaneously.

time slicing A means of scheduling threads or processes wherein each process receives a definite amount of CPU time before returning to the ready queue.

top-down design A method for coding by which the programmer starts with a top-level task and implements subtasks. Each subtask is then subdivided into smaller subtasks. This process is repeated until each remaining subtask is easily coded. *See also* **stepwise refinement** and **structured programming**.

transistor A device with no moving parts that can hold an electromagnetic signal and that is used to build computer circuitry for memory and a processor.

translator A program that converts a program written in one language to an equivalent program in another language.

truth table A means of listing all of the possible values of a Boolean expression.

tuple A linear, immutable collection.

turtle graphics A set of resources that manipulate a pen in a graphics window.

type conversion function A function that takes one type of data as an argument and returns the same data represented in another type.

U

Unified Modeling Language (UML) A graphical notation for describing a software system in various phases of development.

Unicode A character set that uses 16 bits to represent over 65,000 possible characters. These include the ASCII character set as well as symbols and ideograms in many international languages. *See also* **ASCII character set**.

user interface The part of a software system that handles interaction with users.

V

value An item that is associated with a key and is located by a key in a collection.

variable A memory location, referenced by an identifier, whose value can be changed during execution of a program.

variable reference The process whereby the computer looks up and returns the value of a variable.

vector A one-dimensional array that supports resizing, insertions, and removals.

virtual machine A software tool that behaves like a high-level computer.

virtual reality A technology that allows a user to interact with a computer-generated environment, usually simulating movement in three dimensions.

W

waterfall model A series of steps in which a software system trickles down from analysis to design to implementation. *See also* **software development life cycle**.

Web client Software on a user's computer that makes requests for resources from the Web.

Web server Software on a computer that responds to requests for resources and makes them available on the Web.

while loop A pretest loop that examines a Boolean expression before causing a statement to be executed.

window A rectangular area of a computer screen that can contain window objects. Windows typically can be resized, minimized, maximized, zoomed, or closed.

window object (widget) A computational object that displays an image, such as a button or a text field, in a window and supports interaction with the user.

INDEX

Note: **Boldface** type indicates key terms.

Special Characters

- \ (backslash), 49, 50, 60
- < (left angle bracket), 25, 92, 99, 163, 312, 313
- > (right angle bracket), 24, 25, 92, 99, 163, 312, 313
- ! (exclamation mark), 92, 99, 163, 312, 313
- “ (double quotation mark), 50
- % (percent sign), 58, 85, 99, 311
- ‘ (single quotation mark), 50
- * (asterisk), 58, 99, 311
- + (plus sign), 50, 58, 162, 163, 311
- (minus sign), 58, 99, 311
- / (forward slash), 58, 99, 311
- = (equal sign), 92, 97, 99, 162, 163, 171, 312, 313
- [] (square brackets), 123–124, 160, 161, 164
- _ (underscore), 51
- ... (ellipsis), 26

A

- abacus**, 12
- ABC (Atanasoff-Berry Computer), 15, 16
- abstract behavior**, 340
- abstract classes**, 340–341
- abstraction**, 18, 52, 202
- accept** method, 416
- acceptCommand** function, 222
- accessor(s)**, 300
- accessor methods**, 259
- Account** class, 340–341
- acquire** method, 407
- ACTIVE** constant, 384
- __add__** method, 311
- add(account)** method, 318
- addition** operator (+), 58, 99
- aDict** operation, 187
- Advanced Research Projects Agency Network (ARPANET), 21

- Aiken, Howard, 15
- algorithms**, 2–4
 - information processing related, 5
- aliasing**, 169–170
- Al-Khawarizmi, Muhammad ibn Musa, 11
- Allen, Paul, 21
- Altair, 20
- Alto, 20
- analog information**, 267
- analysis phase**, 40, 41. *See also* software development
- and** logical operator, 97–98, 99
- anonymous functions**, 237
- API (application programming interface), image-processing library, 439–440
- append** method, 165, 166, 167
 - finding median of a set of numbers, 172–173
- Apple Computer, 20
 - HyperCard, 22
- application programming interface (API), image-processing library, 439–440
- applications software (applications)**, 9
- approximating square roots case study, 110–113
- _area** method, 366
- arguments**
 - default (keyword), namespace, 230–232
 - functions, 176
- arithmetic**
 - mixed-mode, 60–61
 - rational numbers, 311
- arithmetic expressions**, 58–60
- arithmetic negation** operator (-), 58, 99
- arithmetic operators**, 58, 99
 - overloading, 312
- ARPANET (Advanced Research Projects Agency Network), 21
- artificial intelligence**, 17
- ASCII set**, 55–57
- aspect ratio**, 283
- assemblers**, 17
- assembly languages**, 16–17

- assignment operator (=), 97, 98, 99, 171
- assignment statements**, 51
- association**, 183
- association lists**, 183
- asterisk (*)
 - exponentiation operator (**), 58, 99
 - `__mul__` method, 311
 - multiplication operator (*), 58, 99
- Atanasoff, John, 15
- Atanasoff-Berry Computer (ABC), 15, 16
- ATM case studies, 325–331
 - GUI-based, 367–372
- ATM class, 368–372
- augmented assignment operations**, 79–80

B

- Babbage, Charles, 14
- backslash character (\), 50, 60
- backspace character (\b), 50
- Backus, John, 17
- Bank class, 317–319, 325, 326, 327–331, 368
- base cases**, 211
- base ten number system**, 129. *See also* decimal number system
- base two number system**, 129. *See also* binary number system
- batch processing**, 19
- `begin_fill` method, 250
- benefits, object-oriented programming, 341–343
- Berners-Lee, Tim, 22, 23
- Berry, Clifford, 15
- binary digits**, 7
- binary number system**, 129, 130
 - converting binary to decimal, 131–132, 133–134
 - converting decimal to binary, 132–133
- `bind` method, 289, 387–388, 415
- bit(s)**, 7
- bit strings**, 131
- bit-mapped display screens**, 20
- black and white, converting images to, 276–278
- `blackAndWhite` function, 276–277

- `Blackjack` class, 335–340
- blackjack game, dealer and play in, 276–278
- Block, threads, 399
- block ciphers**, 129
- `blur` function, 280–281
- blurring images**, 280–281
- `<Btn-Motion>` event, 387
- Boole, George, 14–15
- Boolean data type**, 91
- Boolean expressions, compound, 97–99
- Boolean functions**, 177
- bottom-up testing**, 152
- bouncy program, 350
 - GUI-based, 351–353
 - terminal-based, 350–351
- `box.activate(index)` method, 384
- `box.curselection` method, 384
- `box.delete(index)` method, 384
- `box.get(index)` method, 384
- `box.see(index)` method, 384
- `box.size` method, 384
- `box.xview` method, 384
- `box.yview` method, 384
- `break` statements, 105–106
- Bush, Vannevar, 19–20
- Button object, 358
 - `<ButtonPress-n>` event, 387
 - `<ButtonRelease-n>` event, 387
- byte code**, 30
- `bytes` function, 416
- `bytes` object, 416

C

- C++, transition from Python to, 441
- Caesar ciphers**, 128–129
- call stacks**, 216
- Canvas class, 251
- Card class, 321–322, 361
- card readers**, 17
- `CardDemo` class, 361–363
- case studies of software development. *See* software development

- cathode ray tubes (CRTs), 19
- c-curve**, 262–266
- cCurve function, 266
- center method, 138
- central processing unit (CPU)**, 6
 - functions, 7–8
- cfg files, setting up, 260
- changePerson function, 193
- character sets**, 55–57
- chat rooms, multi-client, case study, 423–427
- chat script, two-way, 416–418
- ChatRecord class, 424
- chdir function, 147
- CheckingAccount class, 340–341
- chips, microcomputer, 20
- cipher(s), block, 129
- cipher text**, 128
- circlearea program, 364–365, 375, 377, 378
- class(es), 283–302. *See also* classes listed by name
 - abstract, 340–341
 - accessors, 300
 - data-modeling examples. *See* data modeling
 - defining, rules of thumb for, 302
 - docstrings, 298
 - __init__ method, 299
 - instance variables, 299–300
 - lifetime of objects, 301
 - method definitions, 298
 - mutators, 300
 - overview, 294
 - parent, 296
 - __str__ method, 300
 - structuring with inheritance and polymorphism, 331–344
 - subclasses, 296
- class diagrams**, 326–327
- class hierarchies**, 296
- class variables**, 315–317
- classes listed by name
 - Account class, 340–341
 - ATM class, 368–372
 - Bank class, 317–319, 325, 326, 327–331, 368
 - Blackjack class, 335–340
 - Canvas class, 251
 - Card class, 321–322, 361
 - CardDemo class, 361–363
 - ChatRecord class, 424
 - CheckingAccount class, 340–341
 - ClientHandler class, 418–419, 424, 425–427
 - Condition class, 406–407
 - Consumer class, 404, 405
 - Deck class, 321, 323–324
 - Die class, 304, 305, 306–309
 - Doctor class, 420
 - DoubleVar class, 363
 - Frame class, 356, 368, 373
 - Image class, 439–440
 - IntVar class, 363
 - object class, 296
 - PhotoImage class, 357
 - Player class, 304, 305, 306–309
 - Producer class, 404, 405
 - Rational class, 309–311
 - SavingsAccount class, 315–317, 325, 326, 327–328, 340–341, 368
 - Screen class, 251
 - SharedCell class, 404–405, 406, 408
 - SleepyThread class, 401–402
 - StringVar class, 363
 - Student class. *See* Student class
 - Thread class, 399, 418
- clear operation, 185, 188
- client(s), 411
 - multiple. *See* multiple clients
- ClientHandler class, 418–419, 424, 425–427
- client/server programming, 395–428
 - clients, 411
 - day/time client script, 412–414
 - day/time server script, 414–416
 - history, 396–397
 - IP addresses, 409–411
 - multi-client chat room case study, 423–427
 - multiple concurrent clients, 418–419
 - ports, 411
 - producer/consumer relationship, 402–408
 - servers, 411
 - setting up conversations for others, 420–422
 - sockets, 412–414
 - synchronization, 403–408

- threads, 397–402
 - two-way chat script, 416–418
- clone** method, 271, 279
- close** method, 142, 146, 416
- COBOL** (Common Business Oriented Language), 17
- coding. *See* implementation phase
- color(s)**
 - GUI-based programs, 373
 - random, drawing with, 257–259
 - RGB system, 256–257
- Color** attribute, Turtle graphics, 249
- color palettes**, 269
- Colossus**, 15–16
- columnconfigure** method, 378
- command buttons**, 352, 358–360
- command prompts, running scripts, 68–69
- Common Business Oriented Language** (COBOL), 17
- comparison methods, 312–313
- compilers**, 17
- complexity, hiding of, by functions, 203–204
- components, organizing with nested frames, 380–381
- compound Boolean expressions**, 97–99
- compression, lossless and lossy, 269
- Compute** button, 352, 353, **354**
- computeInterest** method, 315, 318
- computer(s)**
 - electronic, first, 15–16
 - mainframe, 16
 - mechanical, 11–15
 - personal, 19–21
- computer systems. *See also* hardware; software
 - history, 10–23
 - structure, 6–10
- computing agents**, 3
- concatenation, strings, 50
- concatenation operator (+), 50, 162, 163
 - object identity and structural equivalence, 171
- concurrent processing**, 19
- condition(s)**, 91
- Condition** class, 406–407
- conditional iteration, 102–109. *See also* while loops
- connect** method, 413
- constants, symbolic, 51
- constructors**, 252
- Consumer** class, 404, 405
- container object**, 363
- context switches**, 399
- continuation condition**, 102
- continuous ranges of values**, 267
- Control Program for Microcomputers (CP/M)**, 21
- control statements**, 75–115
 - conditional iteration. *See* while loops
 - definite iteration. *See* for loops
 - formatting text for output, 83–86
 - selection, 91–100
- controller**, **354**
- convert** function, 189
- coordinate systems**, 248
 - screen, 270
- copying images, 279
- correct programs**, 46
- costs
 - maintenance, 42
 - object-oriented programming, 341–343
 - recursion, 216–218
 - repairing mistakes, 41–42
- count** method, 138
- count** variable, 105
- countBytes** function, 223
- count-controlled loops**
 - conditional iteration, 104–105
 - definite iteration, 77–79
- countFiles** function, 223
- CP/M** (Control Program for Microcomputers), 21
- CPU**. *See* central processing unit (CPU)
- craps game case study, 303–309
- CRTs** (cathode ray tubes), 19
- customer request phase**, 40, 41. *See also* software development

D

- data**, 4
- data encapsulation**, 331

- data encryption**, 126–129
- data modeling, rational numbers, 309–312
- data sequences, traversing contents, 80–81
- data structure**, 122
- data types**, 47–48
 - numeric, 54–55
- day/time client script, 412–414
- day/time server script, 414–416
- `_deal` method, 361
- Dealer object, 335–340
- decimal notation**, 55
- decimal number system**, 129, 130
 - converting binary to decimal, 131–132, 133–134
 - converting decimal to binary, 132–133
- Deck class, 321, 323–324
- decode function, 413
- decrypt script, 129
- decryption**, 128
- default arguments**, namespace, 230–232
- defining
 - classes, rules of thumb, 302
 - functions. *See* function definitions
 - methods, 298
 - recursive functions, 211–212
 - variables, 51
- defining the variable**, 51
- definite iteration, 76–83. *See also* for loops
 - augmented assignment operations, 79–80
 - count-controlled loops, 77–79
 - counting down, 82–83
 - executing statements a given number of times, 76–77
 - off-by-one error, 80
 - specifying steps in range, 81–82
 - traversing contents of data sequences, 80–81
- deposit method, 315
- `_deposit` method, 328
- design errors**, 46
- design phase**, 40, 41. *See also* software development
- `detectEdges` function, 281–282
- dictionaries**, 159, 183–190
 - accessing values, 185
 - adding keys, 184
 - finding mode of a list of values, 189–190
 - hexadecimal system, 188–189
 - literals, 183–184
 - operations, 187–188
 - removing keys, 186
 - replacing values, 185
 - traversing, 186–188
- Die class, 304, 305, 306–309
- digitizing images, 267, 268
- discrete values**, 267
- display screens, bit-mapped, 20
- displaying images, 357–358
- distributed systems**, 396–397
- `_div__` method, 311
- division of labor, support by functions, 205
- division operator (`/`), 58, 99
- docstrings**, 52
 - classes, 298
- Doctor class, 420
- doctor program, 191–195
 - design, 209–210
- dots per inch (DPI), 282
- double quotation mark character (`\`), 50
- DoubleVar class, 363
- Down attribute, Turtle graphics, 249
- down method, 250
- draw method, 271, 272
- drawing with random colors, 257–259
- `drawLine` function, 265, 266
- `drawPolygon` function, 254
- `drawSquare` function, 251
- Dynabook, 20

E

- Eckert, J. Presper, 15
- edge detection**, 281–282
- Electronic Numerical Integrator and Calculator (ENIAC), 15, 16
- elements**, lists, 160
- ellipsis (`...`), 26
- empty strings**, 49
- encryption**, 128
- END constant, 384

- `end_fill` method, 250
- end-of-line comments**, 53
- `endswith` method, 138
- Engelbart, Douglas, 19, 20
- ENIAC (Electronic Numerical Integrator and Calculator), 15, 16
- Enigma code, 16
- <Enter> event, 387
- entries
 - justifying, 375
 - sizing, 374–375
- entry fields**, 352
 - text input and output, 363–365
- `Entry` object, 363
- `__eq__` method, 313, 314
- equal sign (=)
 - assignment operator (=), 97, 98, 99, 171
 - equals (equality) operator (==), 92, 162, 163, 312, 313
 - greater than or equal operator (>=), 92, 99, 163, 312, 313
 - less than or equal operator (<=), 92, 163, 312, 313
 - not equals operator (!=), 92, 99, 163, 312, 313
- equals (equality) operator (==), 92, 162, 163, 312, 313
- `error(s)`
 - costs of repairing mistakes, 41–42
 - design, 46
 - logic, 46, 80
 - off-by-one, 80
 - semantic, 59
- error messages, syntax errors, 31–32
- escape sequences**, 50
- Ethernet, 21
- Euclid, 12
- event(s)**, 353
 - responding to, 358–360
- event-driven programming**, 353–355
- event-driven software systems**, 353
- exclamation mark (!), not equals operator (!=), 92, 99, 163, 312, 313
- executing actions**, 3
- `exists` function, 147
- expansion weight**, 378

- exponentiation operator (**), 58, 99
- expressions**, 58–62
 - arithmetic, 58–60
 - mixed-mode arithmetic, 60–61
 - spacing within, 60
 - type conversions, 61–62
- `extend` method, 165, 166, 167
- extensions**, 124
- external memory**, 8

F

- `False` Boolean value, 91, 92
- `False` value, 98–99
 - object identity and structural equivalence, 171
- field width**, 84
- file formats
 - images, 268–269
 - text files, 141
- `file` object, 142
- file systems**, 9
 - information gathering from, case study, 219–227
- filename extensions, 124
- `filesys.py` program, 220–227
- `fillcolor` method, 250
- `filter` function, 236
- filtering, 236
- `find` method, 138, 167
- `findFiles` function, 223–224
- first-class data objects**, functions as, 233–234
- Flesch, Rudolf, 148
- Flesch Index**, 148
- Flesch-Kincaid Grade level Formula**, 149
- `float` data type, 48, 55
- `float` function, 27, 28, 61, 62
- floating-point numbers**, 55
- `Font` object, 374
- `for` loops, 144, 162, 163, 164
 - count control, 77–79
 - dictionaries, 186, 187
 - executing a given number of times, 76
 - finding median of a set of numbers, 172
 - `range` function, 80–83

- for method, 145
- form fillers**, 363
- format operator (%)**, 85
- format strings**, 85
- FORTAN (Formula Translation Language), 17
- forward method, 250
- forward slash (/)
 - `__div__` method, 311
 - quotient operator (`//`), 58
- fractal objects**, 262
 - recursive patterns in fractals case study, 262–266
- frame(s), nested, 380–381
- Frame class, 356, 368, 373
- function(s)**, 64, 201–242. *See also* functions listed by name
 - abstraction mechanism, 202–205
 - anonymous, 237
 - calling, 64–65
 - elimination of redundancy, 202–203
 - hiding of complexity, 203–204
 - higher-order. *See* higher-order functions
 - modules, 63
 - namespace. *See* namespace
 - recursive. *See* recursive functions
 - support for division of labor, 205
 - support of general methods with systematic variations, 204
 - top-down design. *See* top-down design
- function definitions, 175–178
 - arguments, 176
 - Boolean functions, 177
 - main functions, 178
 - parameters, 176
 - return statement, 177
 - syntax, 175–176
- functional programming**, 342
- functions listed by name
 - `acceptCommand` function, 222
 - `blackAndWhite` function, 276–277
 - `blur` function, 280–281
 - `bytes` function, 416
 - `cCurve` function, 266
 - `changePerson` function, 193
 - `chdir` function, 147
 - `convert` function, 189
 - `countBytes` function, 223
 - `countFiles` function, 223
 - `decode` function, 413
 - `detectEdges` function, 281–282
 - `drawLine` function, 265, 266
 - `drawPolygon` function, 254
 - `drawSquare` function, 251
 - `exists` function, 147
 - `filter` function, 236
 - `findFiles` function, 223–224
 - `float` function, 27, 28, 61, 62
 - `getcwd` function, 147
 - `gethostbyname` function, 410
 - `gethostname` function, 410
 - `getsize` function, 147
 - `grayscale` function, 278
 - `help` function, 66, 67
 - `input` function, 26–27, 28
 - `int` function, 27, 28, 61–62, 144
 - `isdir` function, 147
 - `isfile` function, 147
 - `lambda` function, 237–238
 - `len` function, 122, 161, 163
 - `list` function, 161, 163, 187, 188
 - `main` function. *See* main function
 - `map` function, 234–236
 - `max` function, 94, 189
 - `min` function, 94
 - `mkdir` function, 147
 - `nounPhrase` function, 183
 - `odd` function, 177
 - `open` function, 142
 - `playManyGames` function, 306
 - `playOneGame` function, 306
 - `print` function. *See* print function
 - `randint` function, 107–108
 - `randomWalk` function, 255–256
 - `range` function, 80–83, 161
 - `reduce` function, 237
 - `rename` function, 147
 - `reply` function, 192–193, 209–210
 - `repToInt` function, 231

- rmdir function, 147
- round function, 62
- runCommand function, 222, 238–239
- sentence function, 183
- shrink function, 283–284
- socket function, 413
- square function, 175
- str function, 61
- sum function, 212–213

G

garbage collection, 301

Gates, Bill, 21

__ge__ method, 313

general methods, support by functions, 204

generating sentences case study, 179–183

geometry method, 375–376

get method, 185, 363

_get method, 388

get(pin) method, 318

get operation, 187

getAverage method, 296

getBalance method, 315

_getBalance method, 328

getcwd function, 147

getData method, 406

getHeight method, 271

getHighScore method, 296

gethostbyname function, 410

gethostname function, 410

getName method, 296, 315, 399, 400

getPin method, 315

getPixel method, 271, 272

getPoints method, 338

getScore method, 296, 298

getsize function, 147

getValue method, 305

getWidth method, 271

GIF (Graphics Interchange Format), 268

goto method, 250

grammar rules, 179

graphical user interfaces (GUIs), 9, 349–390

- colors, 373

- event-driven programming, 353–355

- grid attributes, 376–379

- GUI-based programs. *See* GUI-based programs

- keyboard events, 388–389

- mouse events, 387–388

- multi-line text widgets, 381–383

- organizing components using nested frames, 380–381

- scrolling list boxes, 384–387

- sizing and justifying an entry, 374–375

- sizing the main window, 375–376

- terminal-based programs, 350–351

- text attributes, 373–374

graphics, 238

- Turtle. *See* Turtle graphics

- vector, 254

Graphics Interchange Format (GIF), 268

grayscale, 278

- converting images to, 278–279

grayscale function, 278

greater than operator (>), 92, 99, 163, 312, 313

greater than or equal operator (>=), 92, 99, 163, 312, 313

greeting method, 420

grid(s), loop pattern for traversing, 274–275

grid attributes, 376–379

grid method, 357, 361

__gt__ method, 313

GUI(s). *See* graphical user interfaces (GUIs)

GUI-based ATM case study, 367–372

GUI-based programs, 351–353, 355–366

- case study of, 367–372

- command buttons, 358–360

- displaying images, 357–358

- entry fields for text input and output, 363–365

- labels, 357

- pop-up dialog boxes, 365–366

- viewing images, 360–363

- windows, 356–357

H

hardware, 6

`has_key` method, 185, 188

Heading attribute, Turtle graphics, 249

`heading` method, 250

`help` function, 66, 67

hexadecimal number system, 129, 130, 135, 188–189

`hideturtle` method, 250

higher-order functions, 233–239

- creating anonymous functions with `lambda`, 237–238
- filtering, 236
- functions as first-class data objects, 233–234
- jump tables, 238–239
- mapping, 234–236
- reducing, 237

high-level programming languages, 9, 10

`hit` method, 338

Hollereith, Herman, 14

`home` method, 250

Homebrew Computer Club, 20

Hopper, Grace Murray, 17

horizontal tab character (`\t`), 50

HTML (Hypertext Markup Language), 23

HTTP (Hypertext Transfer Protocol), 23

HyperCard, 22

hypermedia, 22

Hypertext Markup Language (HTML), 23

Hypertext Transfer Protocol (HTTP), 23

I

IBM (International Business Machines)

- founding, 14

- Microsoft's early partnership with, 21

IBM PC, 21

IDLE

- defining functions, 176

- description, 23–24

- IDEs extending capabilities, 435

- launching, 23, 435

- line breaks, 60

- running, 260–261

- running scripts from within, 28, 29

- threads, 397

- uses, 435

`if` statements, 94–95

- multi-way, 95–96

`if-else` statements, 92–94

`image(s)`

- displaying, 357–358

- processing. *See* image processing; image-processing library

- viewing, 360–363

Image class, 439–440

image processing, 267–284

- analog and digital information, 267–268

- blurring images, 280–281

- converting images to black and white, 276–278

- converting images to grayscale, 278–279

- copying images, 279

- digitizing images, 268

- edge detection, 281–282

- file formats, 268–269

- image properties, 270

- image-manipulation operations, 269–270

- `images` module, 270–274

- loop pattern for traversing a grid, 274–275

- reducing image size, 282–284

- sampling images, 267, 268

- tuples, 275–276

image-processing library

- API, 439–440

`images` library, installing, 437

`images` module, 270–274, 439–440

immutable data structure, 122

imperative programming, 341

implementation phase, 40, 41. *See also* software development

`import` statement, 182

importing resources, modules, 66

`in` operator, 162, 167

- testing for substrings, 125

income tax calculator case study, 43–47

incremental software development, 40

- indefinite iteration, 76
- index(es)**, 123
 - lists, 160
- index method, 167
- indirect recursion**, 215
- infinite loops**, 102
- infinite precision**, 55
- infinite recursion**, 215–216
- information gathering from a file system case study, 219–227
- information processing**, 2, 4–5
 - concurrent, 19
- inheritance**, 331
- inheritance hierarchies**, 332–333
 - `__init__` method, 299, 321, 357, 358, 359, 360, 368, 369–370, 373, 421
- initializing the variable**, 51
- input(s), 5
 - shell, 25
 - text, entry fields, 363–365
- input function, 26–27, 28
- input/output devices**, 6
- insert method, 165, 166, 167
- installing
 - images library, 437
 - Python, 434
- instance(s)**, 252
- instance variables**, 299–300
- instantiation**, 252–254
- int data type, 48, 54
- int function, 27, 28, 61–62, 144
- integers**, 54
- integrated circuits**, 18
- integration phase**, 40, 41
- Intel 8080 processor, 20
- interfaces**. *See also* graphical user interfaces (GUIs); user interfaces
 - classes, 251
 - image-processing library API, 439–440
- internal memory**, 7
- International Business Machines. *See* IBM (International Business Machines)
- Internet, birth, 21
- Internet host**, 411

- interpreters**, 9, 17
 - operation, 29–30
- IntVar class, 363
- invertible matrices**, 129
- investment report case study, 87–90
- IP addresses**, 409–411
- IP names**, 409
- IP numbers**, 409
- is operator, 171
- isAlive method, 400
- isalpha method, 138
- isdigit method, 138
- isdir function, 147
- isdown method, 250
- isfile function, 147
- items**, lists, 160
- items method, 186
- iterations**, 76. *See also* for loops; while loops
 - conditional, 102–109. *See also* while loops
 - definite. *See* definite iteration; for loops
 - indefinite, 76
- iterative software development**, 40

J

- Jacquard, Joseph, loom built by, 13, 14
- Java, transition from Python to, 441
- Jobs, Steve, 20
- join method, 138
- Joint Photographic Experts Group (JPEG) file
 - format, 268
- JPEG (Joint Photographic Experts Group) file
 - format, 268
- jump tables**, 238–239
- justifying entries, 375

K

- Kay, Alan, 20
- Kaypro, 21
- key(s)**, 183
 - adding to dictionaries, 184
 - removing from dictionaries, 186

- keyboard events, 388–389
 - <KeyPress> event, 388
 - <KeyPress-key> event, 388
- keypunch machines, 16
- <KeyRelease> event, 388
- <KeyRelease-key> event, 388
- keyword arguments, namespace, 230–232

L

- label(s), 352
 - GUI-based programs, 357
- Label components, 361
- LabelDemo, 357
- lambda function, 237–238
- __le__ method, 313
- <Leave> event, 387
- left angle bracket (<)
 - less than operator (<), 92, 163, 312, 313
 - less than or equal operator (<=), 92, 163, 312, 313
 - syntax, 25
- left associative operations, 59
- left method, 250
- Leibnitz, Gottfried, 14
- len function, 122, 161, 163, 187
- __len__ method, 324
- length variable, 31
- less than operator (<), 92, 163, 312, 313
- less than or equal operator (<=), 92, 163, 312, 313
- lifetime
 - namespace, 229–230
 - objects, 301
- line breaks, IDLE, 60
- linear loop structure, 274
- LISP (List Processing), 17
- list(s), 81, 159, 160–173
 - aliasing, 169–170
 - association, 183
 - basic operators, 160–163
 - examples, 160
 - finding median of a set of numbers, 172–173

- finding the mode of a list of values, 189–190
- indexes, 160
- lists of lists, 160
- literals, 161
- mutability, 169–170
- searching, 167
- side effects, 169–170
- sorting, 167
- list boxes, scrolling, 384–387
- list function, 161, 163, 187, 188
- list methods, 165–167
- List Processing (LISP), 17
- listen method, 415
- literals, 48
 - dictionaries, 183–184
 - lists, 161
 - string, 48–49
- loaders, 8
- local host, 411
- locks, 406
- logic errors, 46
- logic errors, 80
- logical conjunction operator (and), 97–98, 99
- logical disjunction operator (or), 97, 98, 99
- logical negation, 98
- logical negation operator (not), 98, 99
- logical operators, 97–99
- lookup tables, 188
- loop(s)
 - count-controlled, 77–79
 - counting down, 82–83
 - definite iteration. *See* definite iteration; **for** loops
 - indefinite iteration, 76
 - infinite, 102
 - passes (iterations), 76
- loop body, 76
- loop control variables, 103
- loop headers, 76
- lossless compression, 269
- lossy scheme, 269
- lower method, 138
- __lt__ method, 313
- luminance, 278

M

machine code, 8

Macintosh, first, 20

Macintosh MultiFinder, 396

magnetic storage media, 8

main function, 178, 181, 192, 222, 266, 278, 404

doctor program, 209

sentence-generator program, 207–208

text-analysis program, 206–207

main method, 357

main module, 66–67

mainframe computers, 16

mainloop method, 357

maintenance costs, 42

maintenance phase, 40, 41

map function, 234–236

mapping, 234–236

Mark I, 15

math module, 65–66

matrices, invertible, 129

Mauchly, John, 15

max function, 94, 189

McCarthy, John, 17, 19

median, 172

finding, 172–173

memory, 6

external (secondary), 8

random access (internal; primary), 7

Metcalfe, Bob, 21

method(s). *See also* methods listed by name

definitions, 298

general, support by functions, 204

polymorphic, 340

strings, 136–140

Turtle graphics, 249–251

methods listed by name

accept method, 416

acquire method, 407

__add__ method, 311

add(account) method, 318

append method, 165, 166, 167, 172–173

_area method, 366

begin_fill method, 250

bind method, 289, 387–388, 415

box.activate(index) method, 384

box.curselection method, 384

box.delete(index) method, 384

box.get(index) method, 384

box.size(index) method, 384

box.size method, 384

box.xview method, 384

box.yview method, 384

center method, 138

clone method, 271, 279

close method, 142, 146, 416

columnconfigure method, 378

computeInterest method, 315, 318

connect method, 413

count method, 138

_deal method, 361

_deposit method, 328

deposit method, 315

_div__ method, 311

down method, 250

draw method, 271, 272

end_fill method, 250

endswith method, 138

__eq__ method, 313, 314

extend method, 165, 166, 167

fillcolor method, 250

find method, 138, 167

forward method, 250

__ge__ method, 313

geometry method, 375–376

_get method, 388

get method, 185, 363

get(pin) method, 318

getAverage method, 296

_getBalance method, 328

getBalance method, 315

getData method, 406

getHeight method, 271

getHighScore method, 296

getName method, 296, 315, 399, 400

getPin method, 315

- getPixel method, 271, 272
- getPoints method, 338
- getScore method, 296, 298
- getValue method, 305
- getWidth method, 271
- goto method, 250
- greeting method, 420
- grid method, 357, 361
- __gt__ method, 313
- has_key method, 185, 188
- heading method, 250
- hideturtle method, 250
- hit method, 338
- home method, 250
- index method, 167
- __init__ method, 299, 321, 357, 358, 359, 360, 368, 369–370, 373, 421
- insert method, 165, 166, 167
- isAlive method, 400
- isalpha method, 138
- isdigit method, 138
- isdown method, 250
- items method, 186
- join method, 138
- __le__ method, 313
- left method, 250
- __len__ method, 324
- listen method, 415
- lower method, 138
- __lt__ method, 313
- main method, 357
- mainloop method, 357
- for method, 145
- __mod__ method, 311
- __mul__ method, 311
- __neq__ method, 313
- __new method, 361
- notify method, 407
- notifyAll method, 407
- open method, 146
- pencolor method, 250, 256, 257–258
- play method, 306
- pop method, 165, 166–167, 186
- position method, 250
- __processAccount method, 328
- __quit method, 328
- read method, 143, 146
- readline method, 144, 146
- recv method, 413
- release method, 406, 407
- remove(pin) method, 318
- replace method, 138
- reply method, 420
- resizable method, 376
- right method, 250
- rowconfigure method, 378
- run method, 328, 397, 398, 399, 400, 419
- save method, 271, 274, 319
- send method, 416
- set method, 363
- setData method, 406
- setheading method, 250
- setName method, 399, 400
- setPixel method, 271, 273
- setScore method, 296
- showturtle method, 250
- __shuffle method, 361
- sort method, 167
- split method, 137, 138, 139–140, 145–146, 164
- start method, 398, 399, 400
- startswith method, 138
- __str__ method, 300, 305, 315, 318, 321–322, 340
- strip method, 138
- __sub__ method, 311
- __switch method, 359
- up method, 250
- upper method, 138
- wait method, 406, 407
- width method, 250
- __withdraw method, 328
- withdraw method, 315, 340
- write method, 142, 146
- yview method, 386
- microcomputer chips**, 20
- Microsoft Disk Operating System (MS-DOS), 21
- min function, 94

- minicomputers, 18–19
- minus sign (-)
 - negation operator, 58, 99
 - `_sub_` method, 311
- mixed-mode arithmetic**, 60–61
- `mkdir` function, 147
- `__mod__` method, 311
- mode**, finding the mode of a list of values, 189–190
- mode string**, 142
- model**, 327
- model/view pattern**, 327
- model/view/controller (MVC) pattern**, 354
- module(s)**, 63
 - importing resources, 66
 - importing script as, 67
 - main module, 66–67
- module variables**, namespace, 228
- modulus (%), 58, 99
- Moore's Law**, 18, 19
- mouse events, 387–388
- MS-DOS (Microsoft Disk Operating System), 21
- `__mul__` method, 311
- multi-client chat room case study, 423–427
- multi-line text widgets, 381–383
- multiple clients
 - chats among, 420–422
 - handling concurrently, 418–419
 - multi-client chat room case study, 423–427
- multiplication operator (*), 58, 99
- multiprocessing systems**, 396
- multi-way selection statements, 95–96
- mutability**, 163
- mutator(s)**, 167–168, 300
- mutator methods**, 259
- MVC (model/view/controller) pattern, **354**

N

- names, variables, 51
- namespace**, 227–232
 - default arguments, 230–232
 - lifetime, 229–230

- method names, 228
- module variables, 228
- parameters, 228
- scope, 228–229
- temporary variables, 228
- natural ordering**, 167
- negation operator (-), 58, 99
- `__neq__` method, 313
- nested frames, 380–381
- nested loop structure**, 274–275
- network(s)**, 6
- networked systems**, 396–397
- Neumann, John von, 16
- `_new` method, 361
- newline character (\n), 49, 50
- Newton, Isaac, 14, 110
- NLS (oNLine System) Augment, 20
- nondirective psychotherapy case study, 191–195
- None value, 167, 168
- not equals operator (!=), 92, 99, 163, 312, 313
- not logical operator, 98, 99
- `notify` method, 407
- `notifyAll` method, 407
- `nounPhrase` function, 183
- number(s)**
 - random, 107–108
 - rational, 309–312
 - writing to files, 142–143
- number systems**, 129–135
 - binary (base two), 129, 130
 - converting binary to decimal, 131–132, 133–134
 - converting decimal to binary, 132–133
 - decimal (base ten), 129, 130
 - hexadecimal, 129, 130, 135, 188–189
 - octal, 129, 130, 134–135
 - positional system for representing numbers, 130–131
- numeric data types**, 48, 54–55
 - floating-point numbers, 55
 - integers, 54

O

- object(s), 294
 - inheritance hierarchies, 332–333
 - input, 320
 - lifetime, 301
 - `pickle` for permanent storage, 319–320
- object class, 296
- object identity**, 171
- object-oriented languages**, 294
- object-oriented programming**, 294
 - costs and benefits, 341–343
- octal number system**, 129, 130, 134–135
- odd function, 177
- off-by-one errors**, 80
- one-way selection statements**, 94–95
- oNLine System (NLS) Augment, 20
- open function, 142
- open method, 146
- `operating system.path` module, 222
- operating systems**, 8–9
 - MS-DOS, 21
 - time-sharing, 19, 396
- operator overloading**, 312
- optical storage media**, 8, 21–22
- or logical operator, 97, 98, 99
- order of precedence, 92
- `os` module, 222
- Osborne, 21
- output(s)**, 5
 - formatting text, 83–86
 - shell, 25
 - text, entry fields, 363–365
- overloading arithmetic operators, 312

P

- panes, 380
- Papert, Seymour, 248
- parallel computing**, 397
- parallel systems**, 397

parameters

- functions, 176
- namespace, 228
- parent(s)**, 220
- parent classes**, 296
- parent components**, 357
- Pascal, Blaise, calculator built by, 12, 13–14
- passes**, 76. *See also* conditional iteration; definite iteration; `for` loops; indefinite iteration; iterations; `while` loops
- paths**, 220
- `pencolor` method, 250, 256, 257–258
- percent sign (%)
 - `format` operator (%), 85
 - `__mod__` method, 311
 - remainder operator (modulus), 58, 99
- personal computers, 19–21
- `PhotoImage` class, 357
- `pickle` module, 319–320
- pickling**, 319–320
- pixels**, 256
- pixilation**, 280
- `play` method, 306
- `Player` class, 304, 305, 306–309
- `Player` object, 335–340
- playing cards, 321–324
- playing the game of craps case study, 303–309
- `playManyGames` function, 306
- `playOneGame` function, 306
- plus sign (+)
 - `__add__` method, 311
 - addition operator, 58, 99
 - concatenation operator, 50, 162, 163, 171
- polymorphic methods**, 340
- polymorphism**, 331
- `pop` operation, 187
- `pop` method, 165, 166–167, 186
- ports**, 6, 411
- `position` method, 250
- positional notation**, 130–131
- positional values**, 130
- precedence rules**, 58–59

- <Prefix-Button-n> event, 387
- primary memory**, 7
- print function, 25–26, 28, 162, 163
 - formatting text, 84
 - string literals, 49
- problem decomposition**, 206
- problem instances**, 204
- procedural programming**, 342
- _processAccount method, 328
- processors**, 7. *See also* central processing unit (CPU)
- Producer class, 404, 405
- producer/consumer relationship**, 402–408
- program(s)**, 6
 - correct, 46
 - format, 67
 - GUI-based. *See* GUI-based programs
 - structure, 67–68
 - terminal-based, 350–351
- program comments**, 52–53
 - docstrings, 52
 - end-of-line, 53
- program libraries**, 29
- programming
 - event-driven, 353–355
 - functional, 342
 - imperative, 341
 - object-oriented. *See* object-oriented programming
 - procedural, 342
- programming languages**, 6
 - assembly languages, 16–17
 - first, 16–18
 - high-level, 9, 10
 - strongly typed, 62
- prototypes**, 40, 88
- pseudocode**, 44
- psychotherapy, nondirective, case study, 191–195
- Python
 - installing, 434
 - invention, 23
 - overview, 23
 - transition to Java and C++, 441
- Python Shell window, 24

- Python Virtual Machine (PVM)**, 30
 - threads, 397
- Python Web page, 433

Q

- _quit method, 328
- quotient operator (/), 58

R

- _radiusEntry widget, 289
- RAM (random access memory)**, 7
- randint function, 107–108
- random access memory (RAM)**, 7
- random module, 107
- random numbers**, 107–108
- randomWalk function, 255–256
- range function, 161
 - for loop, 80–83
 - specifying steps in range, 81–82
- Rational class, 309–311
- rational numbers**, 309–312
 - arithmetic, 311
 - operator overloading, 312
- raw image files**, 268
- read method, 143, 146
- readability, text analysis case study, 148–153
- reading
 - numbers from a file, 145–146
 - text from a file, 143–144
- readline method, 144, 146
- ready queue**, 398
- recursion
 - indirect, 215
 - infinite, 215–216
- recursive calls**, 212
- recursive definition**, 214
- recursive design**, 211
- recursive functions**, 211–218
 - constructing using recursive definitions, 214
 - costs and benefits of recursion, 216–218

- defining, 211–212
 - infinite recursion, 215–216
 - tracing, 213
- recursive patterns in fractals case study, 262–266
- recursive steps**, 211
- `recv` method, 413
- `reduce` function, 237
- reducing**, 237
- reducing image size, 282–284
- redundancy, elimination by functions, 202–203
- `release` method, 406, 407
- remainder operator (%), 58, 99
- `remove(pin)` method, 318
- `rename` function, 147
- repetition statements. *See* conditional iteration;
 - definite iteration; `for` loops; indefinite iteration; `loop(s)`; `while` loops
- `replace` method, 138
- replacements dictionary, 192, 193
- `reply` function, 192–193
 - doctor program, 209–210
- `reply` method, 420
- `repToInt` function, 231
- `resizable` method, 376
- resolution**, 282–284
- responsibility-driven design**, 209
- `RestrictedSavingsAccount` subclass, 333–335
- `return` statement, 175, 177, 183
- RGB system**, 256–257
- right angle bracket (>)
 - greater than operator (>), 92, 99, 163, 312, 313
 - greater than or equal operator (>=), 92, 99, 163, 312, 313
 - shell prompt (>>>), 24
 - syntax, 25
- right associative operations**, 59
- `right` method, 250
- `rmdir` function, 147
- root directory**, 220
- root window**, 375
- Rossum, Guido van, 23
- `round` function, 62
- `rowconfigure` method, 378

- row-major traversal**, 275
- `run` method, 328, 397, 398, 399, 400, 419
- `runCommand` function, 222, 238–239
- running scripts, 23–24
 - terminal command prompts, 68–69
- run-time system**, 9, 10
- Russell, Stephen “Slug,” 17

S

- sampling**, 267, 268
- `save` method, 271, 274, 319
- saving, `pickle` for permanent storage of objects, 319–320
- `SavingsAccount` class, 315–317, 325, 326, 327–328, 340–341, 368
- scientific notation**, 55
- scope, namespace, 228–229
- `Screen` class, 251
- screen coordinate systems**, 270
- `Screen` object, 259–260
- scripts**, 28
 - importing as a module, 67
 - running, 23–24, 434–435
 - running from terminal command prompts, 68–69
- scroll bars**, 384
- scrolling list boxes, 384–387
- searching lists, 167
- secondary memory**, 8
- selection statements**, 91–100. *See also* `if` statements; `if-else` statements
 - Boolean data type, 91
 - Boolean expressions. *See* Boolean expressions
 - logical operators, 98
 - multi-way, 95–96
 - one-way, 94–95
 - short-circuit evaluation, 99–100
 - testing, 100
 - two-way, 92–94
- `self` parameter, 298
- `self._name` instance variable, 299–300
- `self._scores` instance variable, 299

- semantic errors**, 59
- semantics**, 59
- semiconductor storage media**, 8
- send** method, 416
- sentence** function, 183
- sentence generation case study, 179–183
- sentence structure, recursion, 214–215
- sentence-generator program, design, 207–209
- sentinels**, 102
- servers, 411
- set** method, 363
- setData** method, 406
- setheading** method, 250
- setName** method, 399, 400
- setPixel** method, 271, 273
- setScore** method, 296
- Shannon, Claude, 15
- SharedCell** class, 404–405, 406, 408
- shell**, 23
 - inputs, 25
 - outputs, 25
 - running code in, 23–25
- shell prompt (>>>), 24
- short-circuit evaluation**, 99–100
- _showOneCard** instance variable, 338
- showturtle** method, 250
- shrink** function, 283–284
- _shuffle** method, 361
- side effects**, lists, 169–170
- single quotation mark character ('), 50
- sizing
 - entries, 374–375
 - main window, 375–376
 - reducing image size, 282–284
- Sleep, threads, 398
- sleeping threads, 400–402
- SleepyThread** class, 401–402
- slicing
 - substrings, 124–125, 165
 - time slicing, 398
- Smalltalk, 20
- sniffing software**, 126
- socket(s)**, 412–414
- socket** function, 413
- socket** module, 410, 412–414, 415
- software**, 6
 - applications, 9
 - operating systems, 8–9, 19, 396
 - system, 8
- software development**, 40–43
 - approximating square roots case study, 110–113
 - ATM case study, 325–331
 - costs, 41–42
 - generating sentences case study, 179–183
 - GUI-based ATM case study, 367–372
 - income tax calculator case study, 43–47
 - incremental and iterative nature of, 40
 - information gathering from a file system case study, 219–227
 - investment report case study, 87–90
 - multi-client chat room case study, 423–427
 - nondirective psychotherapy case study, 191–195
 - playing the game of craps case study, 303–309
 - prototypes, 40
 - recursive patterns in fractals case study, 262–266
 - text analysis case study, 148–153
 - waterfall model, 40, 41
- solid-state devices**, 18
- sort** method, 167
- sorting lists, 167
- source code**, 30
- spacing expressions, 60
- split** method, 137, 138, 139–140, 145–146, 164
- square brackets ([])
 - lists, 160
 - subscript operator, 123–124, 161, 164
- square** function, 175
- square root approximation case study, 110–113
- stack frames**, 216
- start** method, 398, 399, 400
- startswith** method, 138
- states**, 163
- step values**, 82
- stepwise refinement**, 206

- storage media
 - magnetic, 8
 - optical, 8, 21–22
 - semiconductor, 8
- str** function, 61
- __str__** method, 300, 305, 315, 318, 321–322, 340
- string(s)**
 - concatenation, 50
 - construction from numbers and other strings, 62
 - empty, 49
 - format, 85
 - structure, 122
 - substrings. *See* substrings
- string literals, 48–49
- string methods, 136–140
 - list, 138
- StringVar** class, 363
- strip** method, 138
- strongly typed programming languages**, 62
- structural equivalence**, 171
- structure charts**, 206–207
- Student** class, 295–297
 - accessor methods, 300
 - __init__** method, 299
 - lifetime of objects, 301
 - mutator method, 300
 - __sub__** method, 311
- subclass(es)**, 296
- subclass names
 - RestrictedSavingsAccount** subclass, 333–335
- subscript operator ([])**, 123–124, 161, 164
- substrings**, 122
 - slicing, 124–125, 165
 - testing for, with in operator, 125
- subtraction operator (-), 58, 99
- sum function, 212–213
- summations**, 79
- __switch** method, 359
- symbolic constants**, 51
- synchronization problems**, 403–408
- syntax**, 25–26
 - angle brackets (<>), 25
 - function definitions, 175–176

- syntax errors**, 9
 - detecting and correcting, 31–32
- system software**, 8

T

- tables**, 183
- tabular format**, 83–86
- targets**, 164
- temporary variables, namespace, 228
- terminal command prompts
 - launching interactive sessions, 434
 - running scripts, 68–69
- terminal-based interfaces**, 9
- terminal-based programs, 350–351
- termination condition**, 106
- test suites**, 46–47
- testing
 - selection statements, 100
 - for substrings with in operator, 125
- text
 - cipher, 128
 - entry fields for input and output, 363–365
 - formatting for output, 83–86
 - reading from a file, 143–144
 - writing to files, 142
- text analysis case study, 148–153
- text attributes, 373–374
- text editors**, 9, 10
- text files, 141–147
 - accessing and manipulating multiple files and directories on a disk, 146–147
 - format, 141
 - reading numbers from a file, 145–146
 - reading text from a file, 143–144
 - writing numbers to a file, 142–143
 - writing text to a file, 142
- Text** widgets
 - multi-line, 381–383
- text-analysis programs
 - design, 206–207
 - doctor program, 209–210
 - problem decomposition, 206
 - sentence-generator program, 207–209

- thread(s), 397–400
 - sleeping, 400–402
- Thread class, 399, 418
- threading module, 399
- time slicing**, 398
- Time-out, threads, 398
- time-sharing operating systems**, 19, 396
- title bar**, 352
- tkinter component, 355, 363, 373–374
- tkinter library, 439
- tkinter.messagebox component, 355
- top-down design**, 206–210
 - stepwise refinement, 206
 - text-analysis program, 206–207
- tracing recursive functions, 213
- transistors**, 18
- translators**, 9, 10
- traversing
 - dictionaries, 186–188
 - grids, loop pattern for, 274–275
- true color system**, 257
- True value, 91, 92, 98–99
 - object identity and structural equivalence, 171
- truth tables**, 98–99
- try-except statement, 320–321, 410–411
- tuples**, 173, 275–276
- Turing, Alan, 15, 16
- Turtle graphics**, 248–261
 - colors, 256–257
 - coordinate system, 248
 - drawing tow-dimensional shapes, 254–255
 - drawing with random colors, 257–259
 - manipulating screen, 259–260
 - object attributes, 259
 - object instantiation, 252–254
 - operations, 249–251
 - overview, 248–249
 - random walk, 255–256
 - running IDLE, 260–261
 - setting up a cfg file, 260
- turtle module, 252
- two-way chat script, 416–418
- two-way selection statements**, 92–94
- type conversion functions**, 27, 61–62
- type fonts**, 373–374

U

- UML (Unified Modeling Language) diagrams, 327
- underscore (`_`), 51
- Unicode set**, 55
- Unified Modeling Language (UML) diagrams, 327
- up method, 250
- upper method, 138
- user interfaces**, 9
 - GUIs. *See* graphical user interfaces (GUIs)
 - terminal-based, 9

V

- values
 - accessing in dictionaries, 185
 - continuous range of, 267
 - discrete, 267
 - replacing in dictionaries, 184–185
- variable(s)**, 27, 51–52
 - defining (initializing), 51
 - instance, 299–300
 - names, 51
 - purposes, 52
- variable identifiers (variables)**, 27
- variable references**, 51–52
- vector graphics**, 254
- viewing images, 360–363
- virtual machines**, 9
- virtual reality**, 22
- vocabulary**, 179

W

- Wait, threads, 399
- wait method, 406, 407
- WANs (wide area networks), 21
- waterfall model**, 40, 41
- Web browsers**, 23
- Web clients**, 23
- Web servers**, 23
- Weizenbaum, Joseph, 191

- while loops**, 102–109, 112, 164
 - break** statements, 105–106
 - count control, 104–105
 - errors, 108
 - logic, 108
 - operation, 102–104
 - random numbers, 107–108
 - semantics, 102–103
 - structure, 102
 - testing, 108
 - True** Boolean value, 105–106
- while True** loops, 105–106, 108, 112
- wide area networks (WANs), 21
- widgets**, 351
 - text, multi-line, 381–383
- Width attribute
 - Turtle graphics, 249
- width** method, 250
- window(s)**
 - GUI-based programs, 356–357
 - IDLE. *See* IDLE

- window objects**, 351
- withdraw** method, 315, 340
- _withdraw** method, 328
- World Wide Web, 22–23
- write** method, 142, 146
- writing
 - numbers to a file, 142–143
 - text to a file, 142

X

- Xerox, 20

Y

- yview** method, 386