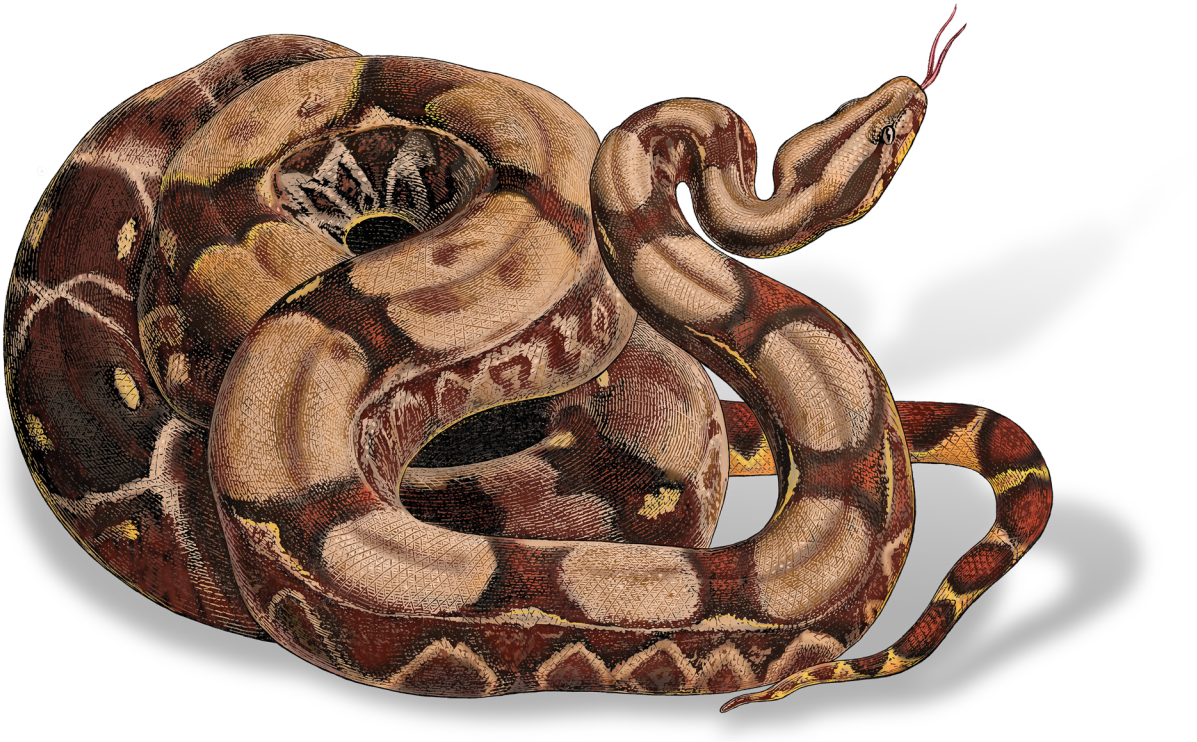
O'REILLY®

Powerful Python

Patterns and Strategies with Modern Python



Aaron Maxwell

"The concepts taught in *Powerful Python* are essential for anyone who takes their Python skills seriously, and the relatable prose and clear examples make it as simple as possible for the reader to learn those concepts."

Rodrigo Girão Serrão
Author of mathspp.com

Powerful Python

How do you become proficient at writing complex, powerful Python applications—without wasting time rehashing the basics you already know or getting bogged down in features that just don't matter? In this unique book, author Aaron Maxwell focuses on the Python first principles that act to accelerate everything else: the 5% of programming knowledge that makes the remaining 95% fall like dominos.

You'll learn:

- Higher-order function abstractions to create powerful, expressive code
- How to make all your Python code more robust and scalable with generator design patterns
- Cognitive benefits of Pythonic comprehensions, how to build more complex comprehension structures, and their surprising link with generators
- Metaprogramming with decorators, for potent abstractions and code reuse patterns that cannot be captured any other way
- Python's exception model for "out of band" signaling of errors and other events
- Advanced object-oriented programming techniques within Python's object model
- How to leverage test-driven development to write better software faster and get into "flow" coding states
- Effective module organization, basic and advanced Python logging, and more

Aaron Maxwell is a software engineer and Pythonista. Through a decade working in Silicon Valley engineering teams, he gained production experience in backend engineering at scale, data science and machine learning, test automation infrastructure, DevOps and SRE, cloud infrastructure, marketing automation, and coding in a variety of languages. He's taught advanced Python to over 10,000 technology professionals worldwide.

PYTHON / PROGRAMMING

US $64.99 CAN $81.99
ISBN: 978-1-098-17570-2

O'REILLY®

Powerful Python

Patterns and Strategies with Modern Python

Aaron Maxwell

O'REILLY®

Powerful Python
by Aaron Maxwell

Printed in the United States of America.

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

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

Acquisitions Editor: Brian Guerin
Development Editor: Virginia Wilson
Production Editor: Aleeya Rahman
Copyeditor: Helena Stirling
Proofreader: Krsta Technology Solutions
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

November 2024: First Edition

Revision History for the First Edition
2024-11-08: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098175702* for release details.

978-1-098-17570-2

[LSI]

Table of Contents

Preface

Python has become the *lingua franca* of modern computing. The thesis of this book is that Python is the most important programming language in the world today…with outsized rewards for those who master it. This book is designed to teach you techniques, patterns, and tools to permanently catapult your skill with everything Python has to offer.

If you write Python code at least part of the time, this book will vastly amplify what you can accomplish and increase the speed at which you do it. And *slash* the amount of time you spend debugging, too.

Who This Book Is For

This book is for you if you know the basics of Python and have mastered just about everything the beginner tutorials can teach you. It is also for those who want to learn more advanced techniques and strategies, so you can do more with Python, and more with coding, than you could before.

This book is *not* for people who want just enough Python to get by. Like I said, Python is important, and rewards those who master it.

And this book is not for the unambitious. In writing, I assume you want to build a career you are proud of, doing work with a high positive impact.

Further, this book is not for the mentally rigid. The difference between elite engineers and "normal" coders lies in the distinctions they make, the mental models they leverage, and their ability to perceive what others cannot.

The Two Levels of Learning

It is not enough to gather knowledge. What you really want is to develop new *capabilities*. Hence, this book recognizes two levels of learning.

The first is the *information level*. This is the level of learning where you read something, or I tell you something, and you memorize it. This puts facts, opinions, and other information in your mind that you can recall later; parrot back to me; and use in logical reasoning.

Which is great. We certainly need this, as a foundation.

But there is a deeper level of learning, called the *ability level*. The ability to do things you could not do before, when you are writing code.

Both are important. But the ability level is what truly matters.

You see, the information level can be deceptive. It makes you *feel* like you understand something. But then you go to write code using it, staring at a blank editor…and you find yourself stuck. "Wait a second. How do I actually use this?"

Know that feeling? Of course you do. Every coder does.

That feeling means you have learned at the information level, but not yet at the ability level. Because when you do, what you need just comes out of you, as naturally as thought itself.

For the most part, reading a book or watching a video can only teach you at the information level. But this book aims to break that trend in several ways.

Our Strategy in This Book

Modern Problem #1: You have too much to learn.

Modern Problem #2: Society has evolved to reduce your time and energy for deep focused learning, due to changes in technology and culture.

This seems like a recipe for misery. But there is a way out: what are called *first principles*.

In any field of human activity—including Python coding—there are foundational concepts which everything builds on. These include powerful distinctions, abstractions, and mental models. When you learn what these first principles are and how to work with them, you find yourself cutting through the noise and getting ahead much more easily.

These first principles are *accelerators*, in that they give you the tools, inner resources, and capabilities to solve many problems. It effectively creates a "95/5" rule: there is a 5% you can focus on learning, which makes the remaining 95% fall like dominos.

That 5% is what we mean by the first principles of Python. Which is what this book is really about.

Hence, this book is *selective* in what it covers. It is not a comprehensive "one stop shop" for everything Python. Further, this book contains *practical* guidance based on lessons learned when writing real-world software—often as part of a team of engineers.

So factors like maintainability, robustness, and readability are considered more important than anything else. There is a balance between leveraging powerful abstractions, and writing code that is easy to work with by everyone on your team. This book aims to walk that line.

Throughout, I give much attention to *cognitive* aspects of software development. How do you write code that you and others can reason about easily, quickly, *and* accurately? This is one reason variable and function naming is important. But it goes far beyond that syntax level… to intelligently choosing which language features and library resources to use, and which to avoid.

This book is not large, as measured by number of pages. That's a feature, not a bug: you already have too much to read. The focus is on what's most valuable, so that—as much as possible—everything you learn will serve you for *years*.

Convention for Callables

This book employs a writing convention that purists will find controversial. In prose, when referring to the names of identifiers, I use a monospace format. So the variable "x" will be `x`, the class named "Point" will be `Point`, and so on.

The impurity: when referring to a function or method, I will append a pair of parentheses to the identifier name. So the function called "compute" is referred to as `compute()`, not `compute`. I do this even when the function must be called with arguments; that "`()`" is essentially an annotation, declaring that this identifier is callable. In my experience teaching and writing about advanced Python, this improves reading comprehension, so I maintain this convention throughout this book.

What's Not Covered

Here are some topics I have chosen to omit:

- I barely mention anything outside the standard library. We have plenty to cover just for Python and its included batteries.
- Type annotations. As we go to press, the dust is still settling on this rich feature. And as dear as it is to some, it is far from universally used.
- Dataclasses. There are endless tutorials on this tool, and Chapter 6, "Classes and Objects: Beyond the Basics" is already the largest in the book.

- Concurrency. The fact is, most Python is written as single-threaded programs. And doing justice to threading, multiprocessing, and asyncio could double the page count.
- Anything depending on specific Python versions. Fortunately, the Python patterns and strategies that work best are surprisingly independent of version. It is these slow changing yet powerful principles we focus on.
- Less commonly used features such as keyword-only and positional-only arguments, conditional (ternary) expressions, pattern matching, and so on. Not to say they are not useful; but better for them to be covered elsewhere.
- And other topics people like, I am sure.

What *is* present covers the important keys of Python, many of which are not new, but are criminally underused and misunderstood, and will be highly valuable for all Pythonistas.

If you simply cannot bear the injustice of this book not covering your favorite Python topic, I can only refer you to what the French poet Paul Valéry said. Which—translated, paraphrased, and shortened—boils down to: "A work of art is never completed, only abandoned."

Such is this book, which I have invested nearly a full decade of my life producing for you. At some point, if it is to be of value to anyone at all, I must ship this thing.

Getting the Most Out of This Book

It is ultimately up to you to transform the information in this book into ability-level learning. And you do that by putting what you read into practice.

To help, I have created coding exercises for every chapter, plus other fun resources—exclusively for readers of this book. To get these along with email notifications of future book releases, go to *https://powerfulpython.com/register* and follow the instructions.

For professional training options, go to *https://powerfulpython.com* and browse the resources there. If you have feedback on this book; corrections; or suggestions for the future, send them to *aaron@powerfulpython.com*.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Powerful Python* by Aaron Maxwell (O'Reilly). Copyright 2025 MigrateUp LLC., 978-1-098-17570-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/powerful_python*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

This book was nearly a decade in the making. And I have *many* to thank.

First, I want to thank the thousands of readers of the earlier, self-published editions of this book—including the hundreds of professional students in Powerful Python Bootcamp. Your many excellent questions and comments—and pointing out bugs!—helped me continually improve the book from day one.

Speaking of which, the O'Reilly team is stellar. If you are an author considering publishing with this amazing group of people, I cannot recommend them enough. I specifically want to thank my development editor, Virginia Wilson; my production editor, Aleeya Rahman; Sarah Grey and Helena Stirling, who together caught more errors than I thought possible; Brian Guerin, for ensuring the project got started in the first place; Yasmina Greco, for wrangling the live O'Reilly training sessions that formed fertile ground for researching this book; and others I am unfairly not mentioning, or who worked behind the scenes.

But the greatest heroes are the technical reviewers. I want to thank Peter Norvig, whose deep feedback on the final self-published version stratospherically elevated this O'Reilly edition; Rodrigo Girão Serrão, whose exceptional expertise in the Python language prevented what would have been many terrible errors; Jess Males, who saved you all from a number of confusingly worded passages and pointed out how to make them comprehensible; and Han Qi, whose formidably sharp mind made it nearly impossible for any bug to escape detection. To all of you, I cannot express enough my gratitude for your help in creating this wonderful book, and making it the best it can be.

CHAPTER 1

Scaling with Generators

This `for` loop seems simple:

```
for item in items:
    do_something_with(item)
```

And yet, miracles hide here. As you probably know, going through a collection one element at a time is called *iteration*. Few understand how Python's iteration system really works and appreciate how deep and well-thought-out it is. This chapter makes you one of those people. You gain the ability to write *highly scalable* Python applications, which can handle ever-larger data sets in performant, memory-efficient ways.

Iteration is also core to one of Python's most powerful tools: the *generator function*. Generator functions are not just a convenient way to create useful iterators. They enable some exquisite patterns of code organization, in a way that—by their very nature—intrinsically encourage excellent coding habits.

This chapter is special, because understanding it threatens to make you a permanently better programmer *in every language*. Mastering Python generators tends to do that, because of the distinctions and insights you gain along the way. Let's dive in.

Iteration in Python

Python has a built-in function called `iter()`. When you pass it a collection, you get back an *iterator object*:

```
>>> numbers = [7, 4, 11, 3]
>>> iter(numbers)
<list_iterator object at 0x10219dc50>
```

An iterator is an object producing the values in a sequence, one at a time:

```
>>> numbers_iter = iter(numbers)
>>> for num in numbers_iter:
...     print(num)
7
4
11
3
```

You don't normally need to use `iter()`. If you instead write `for num in numbers`, what Python effectively does under the hood is call `iter()` on that collection. This happens automatically. Whatever object it gets back is used as the iterator for that `for` loop:

```
# This...
for num in numbers:
    print(num)

# ... is effectively just like this:
numbers_iter = iter(numbers)
for num in numbers_iter:
    print(num)
```

An iterator over a collection is a separate object, with its own identity—which you can verify with `id()`:

```
>>> # id() returns a unique number for each object.
... # Different objects will always have different IDs.
>>> id(numbers)
4330133896
>>> id(numbers_iter)
4330216640
```

How does `iter()` actually get the iterator? It can do this in several ways, but one relies on a *magic method* called `__iter__()`. This is a method any class (including yours) may define, and it is called with no arguments. Each time, it produces a fresh new iterator object object. Lists have this method, for example:

```
>>> numbers.__iter__
<method-wrapper '__iter__' of list object at 0x10130e4c8>
>>> numbers.__iter__()
<list_iterator object at 0x1013180f0>
```

Python makes a distinction between objects which are *iterators*, and objects which are *iterable*. We say an object is *iterable* if and only if you can pass it to `iter()` and get back a ready-to-use iterator. If that object has an `__iter__()` method, `iter()` will call it to get the iterator. Python lists and tuples are iterable. So are strings, which is why you can write `for char in my_str:` to iterate over `my_str`'s characters. Any container you might use in a `for` loop is iterable.

A `for` loop is the most common way to step through a sequence. But sometimes your code needs to step through in a finer-grained way. For this, use the built-in function

`next()`. You normally call it with a single argument, which is an iterator. Each time you call it, `next(my_iterator)` fetches and returns the next element:

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> # Create a fresh iterator...
>>> names_it = iter(names)
>>> next(names_it)
'Tom'
>>> next(names_it)
'Shelly'
>>> next(names_it)
'Garth'
```

What happens if you call `next(names_it)` again? `next()` will raise a special built-in exception, called `StopIteration`:

```
>>> next(names_it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Raising this specific exception is how an iterator signals that the sequence is done. You rarely have to raise or catch this exception yourself, though we'll see some patterns later where it is useful to do so. A good mental model for how a `for` loop works is to imagine it calling `next()` each time through the loop, exiting when `StopIteration` gets raised.

When using `next()` yourself, you can provide a second argument, for the default value. If you do, `next()` will return that instead of raising `StopIteration` at the end:

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> new_names_it = iter(names)
>>> next(new_names_it, "Rick")
'Tom'
>>> next(new_names_it, "Rick")
'Shelly'
>>> next(new_names_it, "Rick")
'Garth'
>>> next(new_names_it, "Rick")
'Rick'
>>> next(new_names_it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(new_names_it, "Jane")
'Jane'
```

Consider a different situation. What if you aren't working with a simple sequence of numbers? What if you are calculating or reading or otherwise obtaining the sequence elements as you go along?

Let's start with a simple example. Suppose you need to write a function creating a list of square numbers which will be processed by other code:

```
def fetch_squares(max_root):
    squares = []
    for n in range(max_root):
        squares.append(n**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

This works. But there are potential problems lurking here. Can you spot any?

Here's one: what if `MAX` is not 5, but 10,000,000? Or 10,000,000,000? Or more? Your memory footprint will be pointlessly dreadful: the code here creates a *massive* list, uses it *once*, then throws it away. On top of that, the consuming `for` loop cannot even *start* until the entire list of squares has been fully calculated. If some poor human is using this program, they'll wonder if it got stuck.

Even worse: What if you are not doing arithmetic to get each element—which is fast and cheap—but making a truly expensive calculation? Or making an API call over the network? Or reading from a database? Your program will be sluggish, perhaps unresponsive, and might even crash with an out-of-memory error. Its users will think you're a terrible programmer.

The solution is to create an iterator to start with, lazily computing each value only when needed. Then each cycle through the loop happens just in time.

So how do you do that? It turns out there are several ways. But the best way is called a *generator function*. And you're going to love it!

Generator Functions

Python provides a tool called the *generator function*, which…well, it's hard to describe everything it gives you in one sentence. Of its many talents, I'll first focus on how it is a *very* useful shortcut for creating iterators.

A generator function looks a lot like a regular function. But instead of saying `return`, it uses a new and different keyword: `yield`. Here's a simple example:

```
def gen_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
```

Use it in a `for` loop like this:

```
>>> for num in gen_nums():
...     print(num)
0
1
2
3
```

Let's go through and understand this. When you call `gen_nums()` like a function, it immediately returns a *generator object*:

```
>>> sequence = gen_nums()
>>> type(sequence)
<class 'generator'>
```

The *generator function* is `gen_nums()`—what we define and then call. A function is a generator function if and only if it uses "yield" instead of "return". The *generator object* is what that generator function returns when called—`sequence`, in this case.

Memorize This Fact

A generator function will *always* return a generator object. *It can never return anything else.*

This generator object is an iterator, which means you can iterate through it using `next()` or a `for` loop:

```
>>> sequence = gen_nums()
>>> next(sequence)
0
>>> next(sequence)
1
>>> next(sequence)
2
>>> next(sequence)
3
>>> next(sequence)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> # Or in a for loop:
... for num in gen_nums():
...     print(num)
0
1
2
3
```

The flow of code works like this: when `next()` is called the first time, or the `for` loop first starts, the body of `gen_nums()` starts executing at the beginning, returning the value to the right of the `yield`.

Advancing next()

So far, this is much like a regular function. But the next time `next()` is called—or, equivalently, the next time through the `for` loop—the function does not start at the beginning again. It starts on the line *after the `yield` statement*. Look at the source of `gen_nums()` again:

```
def gen_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
```

`gen_nums()` is more general than a function or subroutine. It is actually a *coroutine*. You see, a regular function can have several exit points (otherwise known as `return` statements). But it has only one entry point: each time you call a function, it always starts on the first line of the function body.

A coroutine is like a function, except it has several possible *entry* points. It starts with the first line, like a normal function. But when it "returns", the coroutine is not exiting, so much as *pausing*. Subsequent calls with `next()`—or equivalently, the next time through the `for` loop—start at that `yield` statement again, right where it left off; the reentry point is the line after the `yield` statement.

And that's the key: *Each `yield` statement simultaneously defines an exit point, AND a reentry point.*

For generator objects, each time a new value is requested, the flow of control picks up on the line after the `yield` statement. In this case, the next line increments the variable `n`, then continues with the `while` loop.

Notice we do not raise `StopIteration` anywhere in the body of `gen_nums()`. When the function body finally exits—after it exits the `while` loop, in this case—the generator object automatically raises `StopIteration`.

Again: each `yield` statement simultaneously defines an exit point *and* a reentry point. In fact, you can have multiple `yield` statements in a generator:

```
def gen_extra_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
    yield 42 # Second yield
```

Here's the output you will get when you use it:

```
>>> for num in gen_extra_nums():
...     print(num)
0
1
2
3
42
```

The second `yield` is reached after the `while` loop exits. When the function reaches the implicit return at the end, the iteration stops. Reason through the code above, and convince yourself it makes sense.

Converting to a Generator Function

Let's revisit the earlier example of cycling through a sequence of squares. This is how we first did it:

```
def fetch_squares(max_root):
    squares = []
    for num in range(max_root):
        squares.append(num**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

As an exercise, pause here, open up a new Python file, and see if you can write a `gen_squares()` generator function that accomplishes the same thing.

Done? Great. Here's what it looks like:

```
def gen_squares(max_root):
    for num in range(max_root):
        yield num ** 2
```

```
>>> MAX = 5
>>> for square in gen_squares(MAX):
...     print(square)
0
1
4
9
16
```

Notice something important. `gen_squares()` includes the built-in `range()` function. This returns an iterable object. That is important.

Because imagine `range()` returned a list. If that's the case, and `MAX` is huge, that creates a huge list inside your generator function, completely destroying its scalability.

The larger point: Generator functions are *only* as scalable as their *least* scalable line of code. Generator functions *potentially* have a small memory footprint. But only if you code them intelligently. When writing generator functions, watch out for hidden bottlenecks like this.

Do You Need Generators?

Strictly speaking, we don't *need* generator functions for iteration. We just *want* them, because they make useful patterns of scalability far easier.

For example: can you create an iterator without writing a generator function? Yes, you can. For creating a list of square numbers, you can do it like this:

```
class SquaresIterator:
    def __init__(self, max_root_value):
        self.max_root_value = max_root_value
        self.current_root_value = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_root_value >= self.max_root_value:
            raise StopIteration
        square_value = self.current_root_value ** 2
        self.current_root_value += 1
        return square_value

# You can use it like this:
for square in SquaresIterator(5):
    print(square)
```

Each value is obtained by invoking its `__next__()` method, until it raises `StopIteration`. This produces the same output; but take a look at the source for the `SquaresIterator` class and compare it to the source for the generator above. Which is easier to read? Which is easier to maintain? And when requirements change, which is easier to modify without introducing errors? Most people find the generator solution easier and more natural.

```
 Authors often use the word "generator" by itself,
to mean either the generator function, _or_ the generator object
returned when you call it. Typically the writer thinks the intended
meaning is obvious from the context. Sometimes it is, but often
not. Sometimes the writer is not fully clear on the distinction to
begin with. But it is an important distinction to get. Just as there
is a big difference between a function and the value it returns when
you call it, so is there a big difference between the generator
function and the generator object it returns.
```

In your own thought and speech, I encourage you to only use the phrases "generator function" and "generator object", so you are always clear inside yourself, and in your communication. (This also helps your teammates be more clear.) The only exception:

when you truly mean "generator functions and objects", referring to the language feature as a broad concept, then it's okay to just say "generators". I'll lead by example in this book.

Generator Patterns and Scalable Composability

Here's a little generator function:

```
def matching_lines_from_file(path, pattern):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')
```

This function, `matching_lines_from_file()`, demonstrates several important practices for modern Python, and is worth studying. It does simple substring matching on lines of a text file, yielding lines containing that substring.

The first line opens a read-only file object, called `handle`. If you haven't been opening your file objects using `with` statements, start today. The main benefit is that once the `with` block is exited, the file object is automatically closed—even if an exception causes a premature exit. It's similar to:

```
try:
    handle = open(path)
    # read from handle
finally:
    handle.close()
```

(The `try/finally` is explained in Chapter 5.) Next we have `for line in handle`. This useful idiom—which not many people seem to know about—works in a particular way for text files. With each iteration through the `for` loop, a new line of text will be read from the underlying text file, and placed in the `line` variable.

Sometimes people foolishly take another approach, which I have to warn you about:

```
# Don't do this!!
for line in handle.readlines():
```

The `readlines()` (plural) method reads in the *entire file*, parses it into lines, and returns a list of strings—one string per line. By now, you realize how this can destroy scalability.

It bears repeating: a generator function is only as scalable as its *least* scalable line. So code carefully, lest you create some memory bottleneck that renders the generator function pointless.

Another approach you will sometimes see, which *is* scalable, is to use the file object's `.readline()` method (singular), which manually returns lines one at a time:

```
    # .readline() reads and returns a single line of text,
    # or returns the empty string at end-of-file.
    line = handle.readline()
    while line != '':
        # do something with line
        # ...
        # At the end of the loop, read the next line.
        line = handle.readline()
```

But simply writing `for line in handle` is clearer and easier.

Assignment Expressions

A better way to use `.readline()` is with an *assignment expression*:

```
while (line := handle.readline()) != '':
    # do something with line
    # ...
    # No need to call .readline() again here.
```

An assignment *statement*, like `x = y + 1`, does not itself have a value. In other words, you cannot write code like `if (x = y + 1) > 2:` in Python. This was a deliberate choice in the language design, to make bugs like accidentally using `=` instead of `==` impossible.

But sometimes it is useful to assign a value, *and* use that value on the same line. This `while` loop is a prime example. That is why Python added the `:=` operator.

By using `:=`, we only have to write `line := handle.readline()` once instead of `line = handle.readline()` twice. This is not just syntactic convenience; it can often eliminate repetitive code, as demonstrated here, and in some cases even lets us avoid duplicate computations.

This is properly called an *assignment expression*, but many call it the "walrus operator", because it kind of looks like a walrus if you squint and turn your head just right.

After that, it's straightforward: matching lines have any trailing `\n`-character stripped, and are yielded to the consumer. When writing generator functions, ask yourself: "What is the maximum memory footprint of this function, and how can I minimize it?" You can think of scalability as inversely proportional to this footprint. For `matching_lines_from_file()`, it will be about equal to the size of the longest line in the text file. So it is appropriate for the typical human-readable text file, whose lines are small.

(It's also possible to point it to, say, a ten-terabyte text file consisting of exactly one line. If you expect something like *that*, you'll need a different approach.)

Text Lines to Dicts

Now, suppose a log file contains lines like these:

```
WARNING: Disk usage exceeding 85%
DEBUG: User 'tinytim' upgraded to Pro version
INFO: Sent email campaign, completed normally
WARNING: Almost out of beer
```

Say you exercise `matching_lines_from_file()` like so:

```
for line in matching_lines_from_file("log.txt", "WARNING:"):
    print(line)
```

That yields these records:

```
WARNING: Disk usage exceeding 85%
WARNING: Almost out of beer
```

Suppose your application needs that data in dict form:

```
{"level": "WARNING", "message": "Disk usage exceeding 85%"}
{"level": "DEBUG", "message":
    "User 'tinytim' upgraded to Pro version"}
```

We want to scalably transform the records from one form to another—from strings (lines of the log file) to Python dicts. So let's make a new generator function to connect them:

```
def parse_log_records(lines):
    for line in lines:
        level, message = line.split(": ", 1)
        yield {"level": level, "message": message}
```

Now we can connect the two:

```
# log_lines is a generator object
log_lines = matching_lines_from_file("log.txt", "WARNING:")
for record in parse_log_records(log_lines):
    # record is a dict
    print(record)
```

Of course, `parse_log_records()` can be used on its own:

```
with open("log.txt") as handle:
    for record in parse_log_records(handle):
        print(record)
```

`matching_lines_from_file()` and `parse_log_records()` are like building blocks. Properly designed, they can be used to build different data processing streams. I call this *scalable composability*. It goes beyond designing composable functions and types. Ask yourself how you can make the components scalable, *and* whatever is assembled out of them scalable too.

Composable Interfaces

Let's discuss a particular design point. Both `matching_lines_from_file()` and `parse_log_records()` produce an iterator. (Or, more specifically, a generator object.)[1] But they have a discrepancy on the input side: `parse_log_records()` accepts an iterator, but `matching_lines_from_file()` requires a path to a file to read from. This means `matching_lines_from_file()` combines two functions: reading lines of text from a file, then filtering those lines based on some criteria.

Combining functions like this is often what you want in realistic code. But when designing components to flexibly compose together, inconsistent interfaces like this can be limiting. Let's break up the services in `matching_lines_from_file()` into two generator functions:

```
def lines_from_file(path):
    with open(path) as handle:
        for line in handle:
            yield line.rstrip('\n')

def matching_lines(lines, pattern):
    for line in lines:
        if pattern in line:
            yield line
```

You can compose these like so:

```
lines = lines_from_file("log.txt")
matching = matching_lines(lines, 'WARNING:')
for line in matching:
    print(line)
```

Or even redefine `matching_lines_from_file()` in terms of them:

```
def matching_lines_from_file(pattern, path):
    lines = lines_from_file(path)
    matching = matching_lines(lines, pattern)
    for line in matching:
        yield line
```

Conceptually, this factored-out `matching_lines` does a *filtering* operation; all lines are read in, and a subset of them are yielded. `parse_log_records()` is different. One input record (a `str` line) maps to exactly one output record (a `dict`). Mathematically, it's a *mapping* operation. Think of it as a transformer or adapter. `lines_from_file()` is in a third category; instead of taking a stream as input, it takes a completely different parameter. Since it still returns an iterator of records, think of it as a *source*. And a

1 Remember: every generator object is an iterator. But not ever iterator is a generator object.

real program will eventually want to do something with that stream, consuming it without producing another iterator; call that a *sink*.

You need all these pieces to make a working program. When designing a chainable set of generator functions like this—think of it as a toolkit for constructing internal data pipelines—ask yourself whether each component is a sink or a source; whether it does filtering or mapping; or whether it's some combination of these. Just asking yourself this question leads to a more usable, readable, and maintainable codebase. And if you're making a library which others will use, you're more likely to end up with a toolkit so powerfully flexible, people will use it to build programs you never imagined.

Fanning Out

I want you to notice something about `parse_log_records()`. As I said, it fits in the "mapping" category. And notice its mapping is one input (line of text) to one output (dictionary). In other words, each record in the input (a `str`) becomes *exactly one* record in the output (a `dict`).

That isn't always the case. Sometimes, your generator function needs to consume several input records to create one output record. Or the opposite: one input record yields several output records.

Here's an example of the latter. Imagine a text file containing lines in a poem:[2]

```
all night our room was outer-walled with rain
drops fell and flattened on the tin roof
and rang like little disks of metal
```

Let's create a generator function, `words_in_text()`, producing the words one at a time. Here is a first approach:

```
# lines is an iterator of text file lines,
# e.g. returned by lines_from_file()
def words_in_text(lines):
    for line in lines:
        for word in line.split():
            yield word
```

This generator function[3] takes a *fan out* approach. No input records are dropped, which means it doesn't do any filtering; it's still purely in the "mapping" category of generator functions. But the mapping isn't one-to-one. Rather, one input record produces one or more output records. Run the following code:

2 From "Summer Rain" by Amy Lowell, *https://www.poets.org/poetsorg/poem/summer-rain*

3 `line.split()` returns a list of word strings. A lower-memory-footprint approach would be to create an iterator producing one word at a time.

```
poem_lines = lines_from_file("poem.txt")
poem_words = words_in_text(poem_lines)
for word in poem_words:
    print(word)
```

It will produce this output:

```
all
night
our
room
was
outer-walled
...
```

That first input record—"all night our room was outer-walled with rain"—yields eight words (output records). Ignoring any blank lines in the poem, every line of prose will produce at least one word—probably several.

Fanning In

The idea of fanning out is interesting, but simple enough. It's more complex when we go the opposite direction: fanning *in*. That means the generator function consumes more than one input record to produce each output record. Doing this successfully requires awareness of the input's structure, and you'll typically need to encode some simple parsing logic.

Imagine a text file containing residential house sales data. Each record is a set of key-value pairs, one pair per line, with records separated by blank lines:

```
address: 1423 99th Ave
square_feet: 1705
price_usd: 340210

address: 24257 Pueblo Dr
square_feet: 2305
price_usd: 170210

address: 127 Cochran
square_feet: 2068
price_usd: 320500
```

To read this data into a form usable in our code, what we want is a generator function —let's name it `house_records()`—which accepts a sequence of strings (lines) and parses them into convenient dictionaries:

```
>>> lines_of_house_data = lines_from_file("housedata.txt")
>>> houses = house_records(lines_of_house_data)
>>> # Fetch the first record.
... house = next(houses)
>>> house['address']
'1423 99th Ave'
```

```
>>> house = next(houses)
>>> house['address']
'24257 Pueblo Dr'
```

How would you create this? If practical, try it: pause here, open up a code editor, and see if you can implement it.

Okay, time's up. Here is one approach:

```
def house_records(lines):
    record = {}
    for line in lines:
        if line == '':
            yield record
            record = {}
            continue
        key, value = line.split(': ', 1)
        record[key] = value
    yield record
```

Notice where the `yield` keywords appear. The last line of the `for` loop reads individual key-value pairs. An empty record dictionary is populated with data until `lines` produces an empty line. That signals the current record is complete, so it's `yield`-ed, and a new record dictionary created. The end of the very last record in `house data.txt` is signaled not by an empty line, but by the end of the file; that's why we need the final `yield` statement.

As defined, `house_records()` is a bit clunky if we're normally reading from a text file. It makes sense to define a new generator function accepting just the path to the file:

```
def house_records_from_file(path):
    lines = lines_from_file(path)
    for house in house_records(lines):
        yield house

# Then in your program:
for house in house_records_from_file("housedata.txt"):
    print(house["address"])
```

You may have noticed many of these examples have a bit of boilerplate when one generator function internally calls another. The last two lines of `house_records_from_file` say:

```
    for house in house_records(lines):
        yield house
```

Python provides a shortcut to accomplish this in one line, with the `yield from` statement:

```
def house_records_from_file(path):
    lines = lines_from_file(path)
    yield from house_records(lines)
```

Even though "yield from" is two words, semantically it's a single keyword, distinct from `yield`. The `yield from` statement is used specifically in generator functions, when they yield values directly from another generator object (or, equivalently, by calling another generator function). Using it often simplifies your code, as you see in `house_records_from_file()`.

Going back a bit, here's how it works with `matching_lines_from_file()`:

```
def matching_lines_from_file(pattern, path):
    lines = lines_from_file(path)
    yield from matching_lines(lines, pattern)
```

The formal name for what `yield from` does is "delegating to a subgenerator", which instills a deeper connection between the containing and inner generator objects. In particular, generator objects have certain methods—`send()`, `throw()`, and `close()`—for passing information back *into* the context of the running generator function. I won't cover them here, as they are currently not widely used; you can learn more by reading PEPs 342 and 380. If you do use them, `yield from` becomes necessary to enable the flow of information back into the scope of the running coroutine.

Python Is Filled with Iterators

Iteration has snuck into many places in Python. The built-in `range()` returns an iterable:

```
>>> seq = range(3)
>>> type(seq)
<class 'range'>
>>> for n in seq: print(n)
0
1
2
```

The built-in `map`, `filter`, `zip`, and `enumerate` functions all return iterators:

```
>>> numbers = [1, 2, 3]
>>> big_numbers = [100, 200, 300]
>>> def double(n):
...     return 2 * n
>>> def is_even(n):
...     return n % 2 == 0
>>> mapped = map(double, numbers)
>>> mapped
<map object at 0x1013ac518>
>>> for num in mapped: print(num)
2
4
6
```

```
>>> filtered = filter(is_even, numbers)
>>> filtered
<filter object at 0x1013ac668>
>>> for num in filtered: print(num)
2

>>> zipped = zip(numbers, big_numbers)
>>> zipped
<zip object at 0x1013a9608>
>>> for pair in zipped: print(pair)
(1, 100)
(2, 200)
(3, 300)
```

Notice that `mapped` is something called a "map object", rather than a list of the results of the calculation; `filtered` and `zipped` are similar. These are all iterators—giving you all the benefits of iteration, built into the language.

The Iterator Protocol

This optional section explains Python's iterator protocol in formal detail, giving you a precise and low-level understanding of how generators, iterators, and iterables all work. For the day-to-day coding of most programmers, it's not nearly as important as everything else in this chapter. That said, you need this information to implement custom iterable collection types. Personally, I also find knowing the protocol helps me reason through iteration-related issues and edge cases; by knowing these details, I'm able to quickly troubleshoot and fix certain bugs that might otherwise eat up my afternoon.

If this all sounds valuable to you, keep reading; otherwise, feel free to skip to the next chapter. You can always come back to read it later.

As mentioned, Python makes a distinction between *iterators*, versus objects that are *iterable*. The difference is subtle to begin with, and frankly it doesn't help that the two words sound nearly identical. Keep clear in your mind that *iterator* and *iterable* are distinct but related concepts, and the following will be easier to understand.

Informally, an iterator is something you can pass to `next()`, or use exactly once in a `for` loop. More formally, an object in Python is an iterator if it follows the *iterator protocol*. And an object follows the iterator protocol if it meets the following criteria:

- It defines a method named `__next__()`, called with no arguments.
- Each time `__next__()` is called, it produces the next item in the sequence.
- Until all items have been produced. Then, subsequent calls to `__next__()` raise `StopIteration`.

- It also defines a boilerplate method named `__iter__()`, called with no arguments, and returning the same iterator. Its body is literally `return self`.

Any object with these methods can call itself a Python iterator. You are not intended to call the `__next__()` method directly. Instead, you will use the built-in `next()` function.

To understand better, here is how you might write your own `next()` function:

```
_NO_DEFAULT = object()
def next(it, default=_NO_DEFAULT):
    try:
        return it.__next__()
    except StopIteration:
        if default is _NO_DEFAULT:
            raise
        return default
```

(As a side note, notice how this function creates a unique *sentinel value*, `_NO_DEFAULT`, rather than defaulting to a built-in value like `None`. A sentinel value is a value that exists solely for signaling in the algorithm, and is meant to never overlap with a possible value for real data. This way, you can pass any value to `default` that you like without conflict.)

Now, all the above is for the *iterator*. Let's explain the other word, "iterable". Informally, an object is *iterable* if you can use it in a `for` loop. More formally, a Python object is iterable if it meets one of these two criteria:

- It defines a method called `__iter__()`, which creates and returns an iterator over the elements in the container; or
- It defines `__getitem__()`—the magic method for square brackets—and lets you reference `foo[0]`, `foo[1]`, etc., raising an `IndexError` once you go past the last element.[4]

(Notice "iterator" is a noun, while "iterable" is usually an adjective. This can help you remember which is which.)

When implementing your own container type, you probably want to make it iterable, so you and others can use it in a `for` loop. Depending on the nature of the container, it's often easiest to implement the sequence protocol. As an example, consider this `UniqueList` type, which is a kind of hybrid between a list and a set:

4 This is a subset of what is called the *sequence protocol*. A Python object conforms to the sequence protocol if it defines `__getitem__()`, and also `__len__()`, which returns the length of the sequence. But `iter()` is smart enough to work even if only the `__getitem__()` method is defined.

```
class UniqueList:
    def __init__(self, items):
        self.items = []
        for item in items:
            self.append(item)
    def append(self, item):
        if item not in self.items:
            self.items.append(item)
    def __getitem__(self, index):
        return self.items[index]
```

Use it like this:

```
>>> u = UniqueList([3,7,2,9,3,4,2])
>>> u.items
[3, 7, 2, 9, 4]
>>> u[3]
9
>>> u[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in __getitem__
IndexError: list index out of range
```

The `__getitem__()` method implements square-bracket access; basically, Python translates `u[3]` into `u.__getitem__(3)`. We've programmed this object's square brackets to operate much like a normal list, in that the initial element is at index 0, and you get subsequent elements with subsequent integers, not skipping any. And when you go past the end, it raises `IndexError`. If an object has a `__getitem__()` method behaving in just this way, `iter()` knows how to create an iterator over it:

```
>>> u_iter = iter(u)
>>> type(u_iter)
<class 'iterator'>
>>> for num in u_iter:
...     print(num)
3
7
2
9
4
```

Notice we get a lot of this behavior for free, simply because we're using an actual list internally (and thus delegating much of the `__getitem__()` logic to it). That's a clue for you, whenever you make a custom collection that acts like a list—or one of the other standard collection types. If your object internally stores its data in one of the standard data types, you'll often have an easier time mimicking its behavior.

Sometimes you can shortcut even more by inheriting from something which is iterable. Another (and perhaps better[5]) way to implement `UniqueList` is to inherit from `list`:

```
class UniqueList(list):
    def __init__(self, items):
        for item in items:
            self.append(item)
    def append(self, item):
        if item not in self:
            super().append(item)
```

Then you can treat it as you would any Python list, with its full iterable qualities.

Writing a `__getitem__()` method which acts like a list's is one way to make your class iterable. (And optionally adding `__len__()`.) The other involves writing an `__iter__()` method. When called with no arguments, it must return some object which follows the iterator protocol, described above. In the worst case, you'll need to implement something like the `SquaresIterator` class from earlier in this chapter, with its own `__next__()` and `__iter__()` methods. But usually you don't have to work that hard—you can simply return a generator object instead. That means `__iter__()` is a generator function itself, or it internally calls some other generator function, returning its value.

Iterators must always have an `__iter__()` method, as do some iterables. Both are called with no argument, and both return an iterator object. The only difference: the `__iter__()` for the iterator returns its `self`, while an iterable's `__iter__()` will create and return a *new* iterator. And if you call it twice, you get two different iterators.

This similarity is intentional, to simplify control code that can accept either iterators or iterables. Here's the mental model you can safely follow: when Python's runtime encounters a `for` loop, it will start by invoking `iter(sequence)`. This *always* returns an iterator: either `sequence` itself, or (if `sequence` is only iterable) the iterator created by `sequence.__iter__()`.

Iterables are everywhere in Python. Almost all built-in collection types are iterable: `list`, `tuple`, and `set`, and even `dict`. (Though you'll probably want to use `dict.items()`—a simple `for x in some_dict` will iterate just over the keys).

In your own custom collection classes, sometimes the easiest way to implement `__iter__()` actually involves using `iter()`. For instance, this will not work:

5 This inheritance-based version demonstrates an "is-a" relationship, while the previous version a "has-a" relationship. Both will work, but because a `UniqueList` could be considered a kind of list, one can argue that the latter version makes more logical sense, and it will work in more intuitive ways with functions like `isinstance()`.

```
class BrokenInLoops:
    def __init__(self):
        self.items = [7, 3, 9]
    def __iter__(self):
        return self.items
```

If you try it, you get a `TypeError`:

```
>>> items = BrokenInLoops()
>>> for item in items:
...     print(item)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: iter() returned non-iterator of type 'list'
```

It doesn't work because `__iter__()` is supposed to return an iterator, but a list object is *not* an iterator; it is simply *iterable*. You can fix this with a one-line change: use `iter()` to create an iterator object inside of `__iter__()`, and return that object:

```
class WorksInLoops:
    def __init__(self):
        self.items = [7, 3, 9]
    def __iter__(self):
        # This class is identical to BrokenInLoops,
        # except for this next line.
        return iter(self.items)
```

This makes `WorksInLoops` itself iterable, because `__iter__()` returns an actual iterator object—making `WorksInLoops` follow the iterator protocol correctly. That `__iter__()` method generates a fresh iterator each time:

```
>>> items = WorksInLoops()
>>> for item in items:
...     print(item)
7
3
9
>>> for another_item in items:
...     print(another_item)
7
3
9
```

Conclusion

Infusing your Python code with generators has a profound effect. All the code you write becomes more memory-efficient, more responsive, and more robust. Your programs are automatically able to gracefully handle larger input sizes than you anticipated. And this naturally boosts your reputation as someone who consistently creates high-quality software.

CHAPTER 2

Creating Collections with Comprehensions

A *list comprehension* is a high-level, declarative way to create a list. It looks like this:

```
>>> squares = [ n*n for n in range(6) ]
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

This is essentially equivalent to the following:

```
>>> squares = []
>>> for n in range(6):
...     squares.append(n*n)
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

Notice that in the first example, what you type is declaring *what* kind of list you want, while the second is specifying *how* to create it. That's why we say it is high-level and declarative: it's as if you are stating what kind of list you want created, then letting Python figure out how to build it.

Python lets you write other kinds of comprehensions than lists. Here's a simple dictionary comprehension, for example:

```
>>> blocks = { n: "x" * n for n in range(5) }
>>> print(blocks)
{0: '', 1: 'x', 2: 'xx', 3: 'xxx', 4: 'xxxx'}
```

This is equivalent to the following:

```
>>> blocks = dict()
>>> for n in range(5):
...     blocks[n] = "x" * n
>>> print(blocks)
{0: '', 1: 'x', 2: 'xx', 3: 'xxx', 4: 'xxxx'}
```

The main benefits of comprehensions are readability and maintainability. Most people find them *very* readable; even developers encountering a comprehension for the first time will usually find their first guess about what it means to be correct. You can't get more readable than that.

And there is a deeper, cognitive benefit: once you've practiced with comprehensions a bit, you will find you can write them with very little mental effort—keeping more of your attention free for other tasks.

Beyond lists and dictionaries, there are several other forms of comprehension you will learn about in this chapter. As you become comfortable with them, you will find them to be versatile and very Pythonic—meaning, they fit well into many other Python idioms and constructs, lending new expressiveness and elegance to your code.

List Comprehensions

List comprehensions are the most widely used kind of comprehension and are essentially a way to create and populate a list. Their structure looks like this:

```
[ EXPRESSION for VARIABLE in SEQUENCE ]
```

`EXPRESSION` is any Python expression, though in useful comprehensions, the expression often has some variable in it. That variable is stated in the `VARIABLE` field. `SEQUENCE` defines the source values the variable enumerates through, creating the final sequence of calculated values.

Here's the simple example we glimpsed earlier:

```
>>> squares = [ n*n for n in range(6) ]
>>> type(squares)
<class 'list'>
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

Notice the result is just a regular list. In `squares`, the expression is `n*n`; the variable is `n`; and the source sequence is `range(6)`. The sequence is a `range` object; in fact, it can be any iterable…another list or tuple, a generator object, or something else.

The expression part can be anything that reduces to a value, including:

- Arithmetic expressions like `n+3`
- A function call like `f(m)`, using `m` as the variable
- A slice operation (like `s[::-1]`, to reverse a string)
- Method calls

Some complete examples:

```
>>> # First define some source sequences...
>>> pets = ["dog", "parakeet", "cat", "llama"]
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> # And a helper function...
>>> def repeat(s):
...     return s + s
...
>>> # Now, some list comprehensions:
>>> [ 2*m+3 for m in range(10, 20, 2) ]
[23, 27, 31, 35, 39]
>>> [ abs(num) for num in numbers ]
[9, 1, 4, 20, 11, 3]
>>> [ 10 - x for x in numbers ]
[1, 11, 14, -10, -1, 13]
>>> [ pet.upper() for pet in pets ]
['DOG', 'PARAKEET', 'CAT', 'LLAMA']
>>> [ "The " + pet for pet in sorted(pets) ]
['The cat', 'The dog', 'The llama', 'The parakeet']
>>> [ repeat(pet) for pet in pets ]
['dogdog', 'parakeetparakeet', 'catcat', 'llamallama']
```

Notice how all these fit the same structure. They all have the keywords `for` and `in`; those are required in Python for any kind of comprehension you may write. These are interleaved among three fields: the expression, the variable (the identifier from which the expression is composed), and the source sequence.

The order of elements in the final list is determined by the order of the source sequence. You can filter out elements by adding an `if` clause:

```
>>> def is_palindrome(s):
...     return s == s[::-1]
...
>>> pets = ["dog", "parakeet", "cat", "llama"]
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> words = ["bib", "bias", "dad", "eye", "deed", "tooth"]
>>>
>>> [ n*2 for n in numbers if n % 2 == 0 ]
[-8, 40]
>>>
>>> [pet.upper() for pet in pets if len(pet) == 3]
['DOG', 'CAT']
>>>
>>> [n for n in numbers if n > 0]
[9, 20, 11]
>>>
>>> [word for word in words if is_palindrome(word)]
['bib', 'dad', 'eye', 'deed']
```

The structure is

```
[ EXPR for VAR in SEQUENCE if CONDITION ]
```

where `CONDITION` is an expression that evaluates to `True` or `False`, depending on the variable.[1] Note that it can be either a function applied to the variable (`is_palindrome(word)`), or a more complex expression. Choosing to use a function can improve readability, and also let you apply filter logic whose code won't fit on one line.

A list comprehension must always have the `for` keyword, even if the beginning expression is just the variable itself. For example:

```
>>> [word for word in words if is_palindrome(word)]
['bib', 'dad', 'eye', 'deed']
```

Sometimes people think `word for word in words` seems redundant (because it is), and try to shorten it. But that does not work:

```
>>> [word in words if is_palindrome(word)]
  File "<stdin>", line 1
    [word in words if is_palindrome(word)]
                                        ^
SyntaxError: invalid syntax
```

Formatting for Readability (and More)

Realistic list comprehensions tend to be too long to fit nicely on a single line. And they are composed of distinct logical parts, which can vary independently as the code evolves. This creates a couple of inconveniences, which are solved by a convenient fact: *Python's normal rules of whitespace are suspended inside the square brackets*. You can exploit this to make them more readable and maintainable, splitting them across multiple lines:

```
def double_short_words(words):
    return [ word + word
             for word in words
             if len(word) < 5 ]
```

Another variation, which some people prefer:

```
def double_short_words(words):
    return [
        word + word
        for word in words
        if len(word) < 5
    ]
```

What I've done here is split the comprehension across separate lines. You can, and should, do this with any substantial comprehension. It's great for several reasons, the

1 Technically, the condition does not have to depend on the variable. But useful examples of this are extremely rare.

most important being the instant gain in readability. This comprehension has three separate ideas expressed inside the square brackets: the expression (`word + word`); the sequence (`for word in words`); and the filtering clause (`if len(word) < 5`). These are logically separate aspects, and splitting them across different lines takes less cognitive effort for a human to read and understand than the one-line version. It's effectively preparsed for you, as you read the code.

Splitting a comprehension over several lines has another benefit: it makes version control and code review diffs more pinpointed. Imagine you and I are on the same development team, working on this code base in different feature branches. In my branch, I change the expression to `word * 2`; in yours, you change the threshold to `len(word) < 7`. If the comprehension is on one line, version control tools will perceive this as a merge conflict, and whoever merges last will have to manually fix it.[2] But since this list comprehension is split across three lines, our source control tool can automatically merge both our branches. And if we're doing code reviews like we should be, the reviewer can identify the precise change immediately, without having to scan the line and think.

Multiple Sources and Filters

You can have several `for VAR in SEQUENCE` clauses. This lets you construct lists based on pairs, triplets, etc., from two or more source sequences:

```
>>> colors = ["orange", "purple", "pink"]
>>> toys = ["bike", "basketball", "skateboard", "doll"]
>>>
>>> [ color + " " + toy
...   for color in colors
...   for toy in toys ]
['orange bike', 'orange basketball', 'orange skateboard',
 'orange doll', 'purple bike', 'purple basketball',
 'purple skateboard', 'purple doll', 'pink bike',
 'pink basketball', 'pink skateboard', 'pink doll']
```

Every pair from the two sources, `colors` and `toys`, is used to calculate a value in the final list. That final list has 12 elements, the product of the lengths of the 2 source lists.

Notice the two `for` clauses are independent of each other; `colors` and `toys` are two unrelated lists. Using multiple `for` clauses can sometimes take a different form, where they are more interdependent. Consider this example:

2 I like to think future version control tools will automatically resolve this kind of situation. I believe it will require the tool to have knowledge of the language grammar, so it can parse and reason about different clauses in a line of code.

```
>>> ranges = [range(1,7), range(4,12,3), range(-5,9,4)]
>>> [ float(num)
...   for subrange in ranges
...   for num in subrange ]
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 4.0, 7.0, 10.0, -5.0,
-1.0, 3.0, 7.0]
```

The source sequence—`ranges`—is a list of range objects.[3] Now, this list comprehension has two `for` clauses again. But notice one depends on the other. The source of the second is the variable for the first!

It's not like the colorful-toys example, whose `for` clauses are independent of each other. When chained together this way, order matters:

```
>>> [ float(num)
...   for num in subrange
...   for subrange in ranges ]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'subrange' is not defined
```

Python parses the list comprehension from left to right. If the first clause is `for num in subrange`, at that point `subrange` is not defined. So you have to put `for subrange in ranges` *first*. You can chain more than two `for` clauses together like this; the first clause will just need to reference a previously-defined source, and the others can use sources defined in the previous `for` clause, like `subrange` is defined.

Independent Clauses

Now, that's for chained `for` clauses. If the clauses are independent, does the order matter at all? It does, just in a different way. What's the difference between these two list comprehensions?

```
>>> colors = ["orange", "purple", "pink"]
>>> toys = ["bike", "basketball", "skateboard", "doll"]
>>>
>>> [ color + " " + toy
...   for color in colors
...   for toy in toys ]
['orange bike', 'orange basketball', 'orange skateboard',
'orange doll', 'purple bike', 'purple basketball',
'purple skateboard', 'purple doll', 'pink bike',
'pink basketball', 'pink skateboard', 'pink doll']
>>>
```

3 Refresher: The `range()` built-in returns an iterator over a sequence of integers, and can be called with 1, 2, or 3 arguments. The most general form is `range(start, stop, step)`, beginning at `start`, going up to *but not including* `stop`, in increments of `step`. Called with two arguments, the step-size defaults to 1; with one argument, that argument is the `stop`, and the sequence starts at 0.

```
>>> [ color + " " + toy
...   for toy in toys
...   for color in colors ]
['orange bike', 'purple bike', 'pink bike', 'orange
basketball', 'purple basketball', 'pink basketball',
'orange skateboard', 'purple skateboard', 'pink
skateboard', 'orange doll', 'purple doll', 'pink doll']
```

The order here doesn't matter in the sense it does for chained `for` clauses, where you *must* put things in a certain order, or your program won't run. Here, you have a choice. And that choice *does* affect the order of elements in the final comprehension.

For both versions, the first element is "orange bike". But the second element is different. Ask yourself: why? Why is the first element the same in both comprehensions? And why is the second element different?

It has to do with which sequence is held constant while the other varies. It's the same logic that applies when nesting regular `for` loops:

```
>>> # Nested one way...
... build_colors_toys = []
>>> for color in colors:
...     for toy in toys:
...         build_colors_toys.append(color + " " + toy)
>>> build_colors_toys[0]
'orange bike'
>>> build_colors_toys[1]
'orange basketball'
>>>
>>> # And nested the other way.
... build_toys_colors = []
>>> for toy in toys:
...     for color in colors:
...         build_toys_colors.append(color + " " + toy)
>>> build_toys_colors[0]
'orange bike'
>>> build_toys_colors[1]
'purple bike'
```

The second `for` clause in the list comprehension corresponds to the inner `for` loop. Its values vary through their range more rapidly than those in the outer one.

Multiple Filters

In addition to using several `for` clauses, you can have more than one `if` clause, for multiple levels of filtering. Just write several of them in sequence:

```
>>> numbers = [ 9, -1, -4, 20, 17, -3 ]
>>> odd_positives = [
...     num for num in numbers
...     if num > 0
...     if num % 2 == 1
```

```
... ]
>>> print(odd_positives)
[9, 17]
```

Here, I've placed each `if` clause on its own line, for readability—but I could have put both on one line. When you have more than one `if` clause, they are "and-ed" together, not "or-ed" together. Equivalent to this:

```
>>> numbers = [ 9, -1, -4, 20, 17, -3 ]
>>> odd_positives = [
...     num for num in numbers
...     if num > 0 and num % 2 == 1
... ]
>>> print(odd_positives)
[9, 17]
```

The only difference is readability. When you feel one `if` clause with an "and" will be more readable, do that; when you feel multiple `if` clauses will be more readable, do that.

What if you want to include elements matching *at least one* of the `if`-clause criteria, omitting only those not matching any? In that case, you must use a single `if` clause with an "or". You cannot "or" multiple `if` clauses together inside a comprehension. For example, here's how you can filter based on whether the number is a multiple of 2 or 3:

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> [ num for num in numbers
...   if num % 2 == 0 or num % 3 == 0 ]
[9, -4, 20, -3]
```

You can also define a helper function. When your filtering logic is complex or non-obvious, this will often improve readability, and is worth considering:

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> def num_is_valid(num):
...     return num % 2 == 0 or num % 3 == 0
...
>>> [ num for num in numbers
...   if num_is_valid(num) ]
[9, -4, 20, -3]
```

The comprehension mini-language is not as expressive as Python itself, and some lists cannot be expressed as a comprehension.

You can use multiple `for` and `if` clauses together:

```
>>> weights = [0.2, 0.5, 0.9]
>>> values = [27.5, 13.4]
>>> offsets = [4.3, 7.1, 9.5]
>>>
>>> [ (weight, value, offset)
```

```
...   for weight in weights
...   for value in values
...   for offset in offsets
...   if offset > 5.0
...   if weight * value < offset ]
[(0.2, 27.5, 7.1), (0.2, 27.5, 9.5), (0.2, 13.4, 7.1),
(0.2, 13.4, 9.5), (0.5, 13.4, 7.1), (0.5, 13.4, 9.5)]
```

The only rule is that the first `for` clause must come before the first `if` clause. Other than that, you can interleave `for` and `if` clauses in any order. Most people seem to find it more readable to group all the `for` clauses together at first, then the `if` clauses together at the end.

Comprehensions and Generators

List comprehensions create lists:

```
>>> squares = [ n*n for n in range(6) ]
>>> type(squares)
<class 'list'>
```

When you need a list, that's great, but sometimes you don't *need* a list, and you'd prefer something which does not blow up your memory footprint. It's like the situation near the start of Chapter 1:

```
# This again.
NUM_SQUARES = 10*1000*1000
many_squares = [ n*n for n in range(NUM_SQUARES) ]
for number in many_squares:
    do_something_with(number)
```

The entire `many_squares` list must be fully created—all memory for it must be allocated, and every element calculated—before `do_something_with()` is called even *once*. And memory usage goes through the roof.

You know one solution: write a generator function, and call it. But there's an easier option: write a *generator expression*. This is the official name for it, but it really should be called a "generator comprehension", in my humble but correct opinion. Syntactically, it looks like a list comprehension—except you use parentheses instead of square brackets:

```
>>> generated_squares = ( n*n for n in range(NUM_SQUARES) )
>>> type(generated_squares)
<class 'generator'>
```

This generator expression creates a *generator object*, in the exact same way a list comprehension creates a list. Any list comprehension you write, you can use to create an equivalent generator object, just by swapping "`("and")`" for "` ["and"]`".

And you're creating the object directly, without having to define a generator function to call. In other words, a generator expression is a convenient shortcut when you need a quick generator object:

```
# This...
many_squares = ( n*n for n in range(NUM_SQUARES) )

# ... is EXACTLY EQUIVALENT to this:
def gen_many_squares(limit):
    for n in range(limit):
        yield n * n
many_squares = gen_many_squares(NUM_SQUARES)
```

As far as Python is concerned, these two versions of `many_squares` are completely equivalent.

Everything you know about list comprehensions applies to generator expressions: multiple `for` clauses, `if` clauses, etc. You only need to type the parentheses.

In fact, sometimes you can even omit them. When passing a generator expression as an argument to a function, you will sometimes find yourself typing `((` followed by `))`. In that situation, Python lets you omit the inner pair.

Imagine, for example, you are sorting a list of customer email addresses, looking at only those customers whose status is "active":

```
>>> # User is a class with "email" and "is_active" fields.
... # all_users is a list of User objects.

>>> # Sorted list of active user's email addresses.
... # Passing in a generator expression.
>>> sorted((user.email for user in all_users
...          if user.is_active))
['fred@a.com', 'sandy@f.net', 'tim@d.com']
>>>
>>> # Omitting the inner parentheses.
... # Still passing in a generator expression!
>>> sorted(user.email for user in all_users
...        if user.is_active)
['fred@a.com', 'sandy@f.net', 'tim@d.com']
```

Notice how readable and natural this is (or will be, once you've practiced a bit). One thing to watch out for: you can only inline a generator expression this way when passing it to a function or method of one argument. Otherwise, you get a syntax error:

```
>>>
>>> # Reverse that list. Whoops...
... sorted(user.email for user in all_users
...         if user.is_active, reverse=True)
  File "<stdin>", line 2
SyntaxError: Generator expression must be parenthesized if not sole argument
```

Python cannot interpret what you mean here, because it is ambiguous in Python's grammar. So you must use the inner parentheses:

```
>>> # Okay, THIS will get the reversed list.
... sorted((user.email for user in all_users
...         if user.is_active), reverse=True)
['tim@d.com', 'sandy@f.net', 'fred@a.com']
```

Sometimes it is more readable to assign the generator expression to a variable:

```
>>> active_emails = (
...         user.email for user in all_users
...         if user.is_active
... )

>>> sorted(active_emails, reverse=True)
['tim@d.com', 'sandy@f.net', 'fred@a.com']
```

Generator expressions without parentheses suggest a unified way of thinking about comprehensions, that links generator expressions and list comprehensions together. Here's a generator expression for a sequence of squares:

```
( n**2 for n in range(10) )
```

Here it is again, passed to the built-in `list()` function:

```
list( n**2 for n in range(10) )
```

And here it is as a list comprehension:

```
[ n**2 for n in range(10) ]
```

When you understand generator expressions, it's easy to see list comprehensions as a derivative data structure. The same applies for dictionary and set comprehensions (covered next). Even though Python does not work that way internally, this mental model is fully consistent with Python's semantics.

With this insight, you start seeing new opportunities to use all these comprehension forms in your own code—improving readability, maintainability, and performance in the process.

If generator expressions are so great, why would you ever use list comprehensions? Generally speaking, your code will be more scalable and responsive if you use a generator expression. Except, of course, when you actually need a list. There are several considerations.

First, if the sequence is unlikely to be very big—and by "big", I mean a minimum of thousands of elements long—you probably won't benefit from using a generator expression. That's just not big enough for the memory footprint to matter.

Next, generator expressions do not always fit the usage pattern you need. If you need random access, or to go through the sequence twice, generator expressions won't

work. Generator expressions also won't work if you need to append or remove elements, or change the value at some index so that you can look it up later.

This is especially important when writing a method or function whose return value is a sequence. Do you return a generator expression, or a list comprehension?

In theory, there's no reason to ever return a list instead of a generator object; the caller can turn a generator object into a list just by passing it to `list()`. In practice, the interface may be such that the caller will want an actual list; forcing them to deal with a generator object will just get in the way. Also, if you are constructing the return value as a list within the function, it's silly to return a generator expression over it—just return the actual list.

If your intention is to create a library usable by people who may not be advanced Pythonistas, that can be an argument for returning lists. Almost all programmers are familiar with list-like data structures. But fewer are familiar with how generators work in Python, and may—quite reasonably—get confused when confronted with a generator object.

Dictionaries, Sets, and Tuples

Just like a list comprehension creates a list, a dictionary comprehension creates a dictionary. You saw an example at the beginning of this chapter; here's another. Suppose you have this `Student` class:

```
class Student:
    def __init__(self, name, gpa, major):
        self.name = name
        self.gpa = gpa
        self.major = major
```

Given a list named `students`, containing `Student` instances, we can write a dictionary comprehension mapping student names to their GPAs:

```
>>> { student.name: student.gpa for student in students }
{'Jim Smith': 3.6, 'Ryan Spencer': 3.1,
 'Penny Gilmore': 3.9, 'Alisha Jones': 2.5,
 'Todd Reynolds': 3.4}
```

The syntax differs from that of list comprehensions in two ways. Instead of square brackets, you're using curly braces—which makes sense, since this creates a dictionary. The other difference is the expression field, whose format is "`key: value`", since a `dict` has key-value pairs. So the structure is:

```
{ KEY : VALUE for VARIABLE in SEQUENCE }
```

These are the only differences. *Everything* else you learned about list comprehensions applies, including filtering with `if` clauses:

```
>>> def invert_name(name):
...     first, last = name.split(" ", 1)
...     return last + ", " + first
...
>>> # Get "lastname, firstname" of high-GPA students.
... { invert_name(student.name): student.gpa
...   for student in students
...   if student.gpa > 3.5 }
{'Smith, Jim': 3.6, 'Gilmore, Penny': 3.9}
```

You can create sets too. Set comprehensions look exactly like list comprehensions, but with curly braces instead of square brackets:

```
>>> # A list of student majors...
... [ student.major for student in students ]
['Computer Science', 'Economics', 'Computer Science',
 'Economics', 'Basket Weaving']
>>> # And the same as a set:
... { student.major for student in students }
{'Economics', 'Computer Science', 'Basket Weaving'}
>>> # You can also use the set() built-in.
... set(student.major for student in students)
{'Economics', 'Computer Science', 'Basket Weaving'}
```

(How does Python distinguish between a set and dict comprehension? `dict`'s expression is a key-value pair, while `set`'s is a single value.)

What about tuple comprehensions? This is fun: strictly speaking, Python doesn't support them. However, you can pretend it does by using `tuple()`:

```
>>> tuple(student.gpa for student in students
...       if student.major == "Computer Science")
(3.6, 2.5)
```

This creates a tuple, *but it's not a tuple comprehension*. You're calling the `tuple` constructor, and passing it a single argument. What's that argument? A generator expression! In other words, you're doing this:

```
>>> cs_students = (
...     student.gpa for student in students
...     if student.major == "Computer Science"
...     )
>>> type(cs_students)
<class 'generator'>
>>> tuple(cs_students)
(3.6, 2.5)
>>>
>>> # Same as:
... tuple((student.gpa for student in students
...        if student.major == "Computer Science"))
(3.6, 2.5)
>>> # But you can omit the inner parentheses.
```

`tuple`'s constructor takes an iterator as an argument. The `cs_students` is a generator object (created by the generator expression), and a generator object is an iterator. So you can *pretend* Python has tuple comprehensions, using "`tuple(`" as the opener and "`)`" as the close. In fact, this also gives you alternate ways to create dictionary and set comprehensions:

```
>>> # Same as:
... # { student.name: student.gpa for student in students }
>>> dict((student.name, student.gpa)
...      for student in students)
{'Jim Smith': 3.6, 'Penny Gilmore': 3.9,
 'Alisha Jones': 2.5, 'Ryan Spencer': 3.1,
 'Todd Reynolds': 3.4}
>>> # Same as:
... # { student.major for student in students }
>>> set(student.major for student in students)
{'Computer Science', 'Basket Weaving', 'Economics'}
```

Remember, when you pass a generator expression into a function, you can omit the inner parentheses. That's why you can, for example, type

```
tuple(f(x) for x in numbers)
```

Instead of

```
tuple((f(x) for x in numbers))
```

One last point. Generator expressions are a scalable analog of list comprehensions; is there any such equivalent for dicts, or for sets? No, but you can still construct generator expressions and pass the resulting generator object to their constructor, much like you did with `tuple`.

For dict, you will want the yielded elements to be (key, value) tuples. For sets, it is maximally efficient to code that generator expression to only yield unique values. But that is not always worth the trouble; if duplicates are generated, the set constructor will handle it fine.

Conclusion

Comprehensions are a useful tool for readable, maintainable Python. Their sensible succinctness and high-level, declarative nature make them easy to write, easy to read, and easy to maintain. Use them more in your code, and you will find your Python experience greatly improved.

CHAPTER 3

Advanced Functions

In this chapter, we go beyond the basics of using functions. I'll assume you can write functions with default argument values:

```
>>> def foo(a, b, x=3, y=2):
...     return (a+b)/(x+y)
...
>>> foo(5, 0)
1.0
>>> foo(10, 2, y=3)
2.0
>>> foo(b=4, x=8, a=1)
0.5
```

Notice the way `foo()` is called the last time, with arguments out of order, and everything specified by key-value pairs. Not everyone knows that you can call most Python functions this way. So long as the value of each argument is unambiguously specified, Python doesn't care how you call the function (and this case, we specify `b`, `x`, and `a` out of order, letting `y` be its default value). We will leverage this flexibility later.

This chapter's topics are useful and valuable on their own. And they are important building blocks for some *extremely* powerful patterns, which you will learn in later chapters. Let's get started!

Accepting and Passing Variable Arguments

The `foo()` function above can be called with two, three, or four arguments. Sometimes you want to define a function that can take *any* number of arguments—zero or more, in other words. In Python, it looks like this:

```
# Note the asterisk. That's the magic part
def takes_any_args(*args):
    print("Type of args: " + str(type(args)))
    print("Value of args: " + str(args))
```

Look carefully at the syntax here. `takes_any_args()` is just like a regular function, except you put an asterisk right before the argument `args`. Within the function, `args` is a tuple:

```
>>> takes_any_args("x", "y", "z")
Type of args: <class 'tuple'>
Value of args: ('x', 'y', 'z')
>>> takes_any_args(1)
Type of args: <class 'tuple'>
Value of args: (1,)
>>> takes_any_args()
Type of args: <class 'tuple'>
Value of args: ()
>>> takes_any_args(5, 4, 3, 2, 1)
Type of args: <class 'tuple'>
Value of args: (5, 4, 3, 2, 1)
>>> takes_any_args(["first", "list"], ["another","list"])
Type of args: <class 'tuple'>
Value of args: (['first', 'list'], ['another', 'list'])
```

`args` is a tuple, composed of those arguments passed, in order. Which means if you call the function with no arguments, `args` will be an empty tuple.

This is different from declaring a function that takes a single argument, which happens to be of type list or tuple:

```
>>> def takes_a_list(items):
...     print("Type of items: " + str(type(items)))
...     print("Value of items: " + str(items))
...
>>> takes_a_list(["x", "y", "z"])
Type of items: <class 'list'>
Value of items: ['x', 'y', 'z']
>>> takes_any_args(["x", "y", "z"])
Type of args: <class 'tuple'>
Value of args: (['x', 'y', 'z'],)
```

In these calls to `takes_a_list()` and `takes_any_args()`, the argument `items` is a list of strings. We're calling both functions the exact same way, but what happens inside each function is different. Within `takes_any_args()`, the tuple named `args` has one element—and that element is the list `["x", "y", "z"]`. But in `takes_a_list()`, `items` is the list itself.

This `*args` idiom gives you some *very* helpful programming patterns. You can work with arguments as an abstract sequence, while providing a potentially more natural interface for whomever calls the function.

Above, I always name the argument `args` in the function signature. Writing `*args` is a well-followed convention, but you can choose a different name—the asterisk is what makes it a variable argument. For instance, this takes paths of several files as arguments:

```
def read_files(*paths):
    data = ""
    for path in paths:
        with open(path) as handle:
            data += handle.read()
    return data
```

Most Python programmers use `*args` unless there is a reason to name that variable something else.[1] That reason is usually readability; `read_files()` is a good example. If naming it something other than `args` makes the code more understandable, do it.

Argument Unpacking

The star modifier works in the other direction too. Intriguingly, you can use it with *any* function. For example, suppose a library provides this function:

```
def order_book(title, author, isbn):
    """
    Place an order for a book.
    """
    print(f"Ordering '{title}' by {author} ({isbn})")
    # ...
```

Notice there's no asterisk. Suppose in another, completely different library, you fetch the book info from this function:

```
def get_required_textbook(class_id):
    """
    Returns a tuple (title, author, ISBN)
    """
    # ...
```

Again, no asterisk. Now, one way you can bridge these two functions is to store the tuple result from `get_required_textbook()`, then unpack it element by element:

```
>>> book_info = get_required_textbook(4242)
>>> order_book(book_info[0], book_info[1], book_info[2])
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

1 This seems to be deeply ingrained; once I abbreviated it `*a`, only to have my code reviewer demand I change it to `*args`. They wouldn't approve my pull request until I changed it, so I did.

Writing code this way is tedious and error-prone. Fortunately, Python provides a better way. Let's look at a different function:

```
def normal_function(a, b, c):
    print(f"a: {a} b: {b} c: {c}")
```

No trick here—it really is a normal, boring function, taking three arguments. If we have those three arguments as a list or tuple, Python can automatically "unpack" them for us. We just need to pass in that collection, prefixed with an asterisk:

```
>>> numbers = (7, 5, 3)
>>> normal_function(*numbers)
a: 7 b: 5 c: 3
```

Again, `normal_function` is just a regular function. We did not use an asterisk on the `def` line. But when we call it, we take a tuple called `numbers`, and pass it in with the asterisk in front. This is then unpacked *within the function* to the arguments `a`, `b`, and `c`.

There is a duality here. We can use the asterisk syntax both in *defining* a function, and in *calling* a function. The syntax looks very similar. But realize they are doing two different things. One is packing arguments into a tuple automatically—called *variable arguments*; the other is *un*-packing them—called *argument unpacking*. Be clear on the distinction between the two in your mind.

Armed with this complete understanding, we can bridge the two book functions in a much better way:

```
>>> book_info = get_required_textbook(4242)
>>> order_book(*book_info)
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

This is more concise (less tedious to type), and more maintainable. As you get used to the concept, you'll find it increasingly natural and easy to use in the code you write.

Variable Keyword Arguments

So far we have just looked at functions with *positional* arguments—the kind where you declare a function like `def foo(a, b):`, and then invoke it like `foo(7, 2)`. You know that `a` equal 7 and `b` equals 2 within the function, because of the order of the arguments. Of course, Python also has keyword arguments:

```
>>> def get_rental_cars(size, doors=4,
...         transmission='automatic'):
...     template = "Looking for a {}-door {} car with {} transmission...."
...     print(template.format(doors, size, transmission))
...
>>> get_rental_cars("economy", transmission='manual')
Looking for a 4-door economy car with manual transmission....
```

And remember, Python lets you call functions like this just using keyword arguments:

```
>>> def bar(x, y, z):
...     return x + y * z
...
>>> bar(z=2, y=3, x=4)
10
```

These keyword arguments won't be captured by the `*args` idiom. Instead, Python provides a different syntax—using two asterisks instead of one:

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key} -> {value}")
```

The variable `kwargs` is a *dictionary*. (In contrast to `args`; remember, that was a tuple.) It's just a regular `dict`, so we can iterate through its key-value pairs with `.items()`:

```
>>> print_kwargs(hero="Homer", antihero="Bart", genius="Lisa")
hero -> Homer
antihero -> Bart
genius -> Lisa
```

The arguments to `print_kwargs()` are key-value pairs. This is regular Python syntax for calling functions; what's interesting is happening *inside* the function. There, a variable called `kwargs` is defined. It's a Python dictionary, consisting of the key-value pairs passed in when the function was called.

Here's another example, which has a regular positional argument, followed by arbitrary key-value pairs:

```
def set_config_defaults(config, **kwargs):
    for key, value in kwargs.items():
        # Do not overwrite existing values.
        if key not in config:
            config[key] = value
```

This is perfectly valid. You can define a function that takes some normal arguments, followed by zero or more key-value pairs:

```
>>> config = {"verbosity": 3, "theme": "Blue Steel"}
>>> set_config_defaults(config, bass=11, verbosity=2)
>>> config
{'verbosity': 3, 'theme': 'Blue Steel', 'bass': 11}
```

Like with `*args`, naming this variable `kwargs` is just a strong convention; you can choose a different name if that improves readability.

Keyword Unpacking

Just like `*args`, double-star works the other way too. We can take a regular function, and pass it a dictionary using two asterisks:

```
>>> def normal_function(a, b, c):
...     print(f"a: {a} b: {b} c: {c}")
...
>>> numbers = {"a": 7, "b": 5, "c": 3}
>>> normal_function(**numbers)
a: 7 b: 5 c: 3
```

The keys of the dictionary *must* match up with how the function was declared. Otherwise you get an error:

```
>>> bad_numbers = {"a": 7, "b": 5, "z": 3}
>>> normal_function(**bad_numbers)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: normal_function() got an unexpected keyword argument 'z'
```

This is called *keyword argument unpacking*. It works regardless of whether that function has default values for some of its arguments or not. So long as the value of each argument is specified one way or another, you have valid code:

```
>>> def another_function(x, y, z=2):
...     print(f"x: {x} y: {y} z: {z}")
...
>>> all_numbers = {"x": 2, "y": 7, "z": 10}
>>> some_numbers = {"x": 2, "y": 7}
>>> missing_numbers = {"x": 2}
>>> another_function(**all_numbers)
x: 2 y: 7 z: 10
>>> another_function(**some_numbers)
x: 2 y: 7 z: 2
>>> another_function(**missing_numbers)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: another_function() missing 1 required positional argument: 'y'
```

Combining Positional and Keyword Arguments

You can combine the syntax to use both positional and keyword arguments. In a function signature, just separate `*args` and `**kwargs` by a comma:

```
def general_function(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(f"{key} -> {value}")
```

```
>>> general_function("foo", "bar", x=7, y=33)
foo
bar
x -> 7
y -> 33
```

This usage—declaring a function like `def general_function(*args, **kwargs)`—is the most general way to define a function in Python. A function so declared can be called in any way, with any valid combination of keyword and nonkeyword arguments—including no arguments.

Similarly, you can call a function using both—and both will be unpacked:

```
>>> def addup(a, b, c=1, d=2, e=3):
...     return a + b + c + d + e
...
>>> nums = (3, 4)
>>> extras = {"d": 5, "e": 2}
>>> addup(*nums, **extras)
15
```

There's one last point to understand, on argument ordering. When you `def` the function, you specify the arguments in this order:

1. Positional arguments (nonkeyword) arguments
2. The `*args` nonkeyword variable arguments
3. The `**kwargs` keyword variable arguments

You can omit any of these when defining a function. But any that are present *must* be in this order.

```
# All these are valid function definitions.
def combined1(a, b, *args): pass
def combined2(x, y, z, **kwargs): pass
def combined3(*args, **kwargs): pass
def combined4(x, *args): pass
def combined5(u, v, w, *args, **kwargs): pass
```

Violating this order will cause errors:

```
>>> def bad_combo(**kwargs, *args): pass
  File "<stdin>", line 1
    def bad_combo(**kwargs, *args): pass
                            ^
SyntaxError: invalid syntax
```

Functions as Objects

In Python, functions are ordinary objects—just like integers, lists, or instances of a class you create. The implications are profound, letting you do certain *very* useful things with functions. Leveraging this is one of those secrets separating average Python developers from great ones, because of the *extremely* powerful abstractions which follow.

Once you get this, it can change the way you write software forever. In fact, these advanced patterns for using functions in Python largely transfer to other languages you will use in the future.

To explain, let's start by laying out a problem and solution. Imagine you have a list of strings representing numbers:

```
nums = ["12", "7", "30", "14", "3"]
```

Suppose we want to find the biggest integer in this list. The `max()` built-in does not help:

```
>>> max(nums)
'7'
```

This isn't a bug, of course; since the objects in `nums` are strings, `max()` compares each element lexicographically.[2] By that criteria, "7" is greater than "30", for the same reason "g" comes after "ca" alphabetically. Essentially, `max()` is evaluating each element by a different criterion than what we want.

Since `max()`'s algorithm is simple, let's roll our own that compares based on the integer value of the string:

```
def max_by_int_value(items):
    # For simplicity, assume len(items) > 0
    biggest = items[0]
    for item in items[1:]:
        if int(item) > int(biggest):
            biggest = item
    return biggest
```

```
>>> max_by_int_value(nums)
'30'
```

This gives us what we want: it returns the element in the original list which is maximal, as evaluated by our criteria.

Now imagine working with different data where you have different criteria, for example, a list of actual integers:

```
integers = [3, -2, 7, -1, -20]
```

Suppose we want to find the number with the greatest *absolute value*—i.e., distance from zero. That would be –20 here, but standard `max()` won't do that:

```
>>> max(integers)
7
```

2 Meaning: alphabetically, but generalizing beyond the letters of the alphabet.

Again, let's roll our own, using the built-in `abs` function:

```
def max_by_abs(items):
    biggest = items[0]
    for item in items[1:]:
        if abs(item) > abs(biggest):
            biggest = item
    return biggest
```

```
>>> max_by_abs(integers)
-20
```

One more example—a list of dictionary objects:

```
student_joe = {'gpa': 3.7, 'major': 'physics',
               'name': 'Joe Smith'}
student_jane = {'gpa': 3.8, 'major': 'chemistry',
                'name': 'Jane Jones'}
student_zoe = {'gpa': 3.4, 'major': 'literature',
               'name': 'Zoe Fox'}
students = [student_joe, student_jane, student_zoe]
```

Now, what if we want the record of the student with the highest GPA? Here's a suitable max function:

```
def max_by_gpa(items):
    biggest = items[0]
    for item in items[1:]:
        if item["gpa"] > biggest["gpa"]:
            biggest = item
    return biggest
```

```
>>> max_by_gpa(students)
{'name': 'Jane Jones', 'gpa': 3.8, 'major': 'chemistry'}
```

Just one line of code is different between `max_by_int_value()`, `max_by_abs()`, and `max_by_gpa()`: the comparison line. `max_by_int_value()` compares `int(item)` to `int(biggest)`; `max_by_abs()` compares `abs(item)` to `abs(biggest)`; and `max_by_gpa()` compares `item["gpa"]` to `biggest["gpa"]`. Other than that, these `max()` functions are identical.

I don't know about you, but having nearly identical functions like this drives me nuts. The way out is to realize the comparison is based on a value *derived* from the element—not the value of the element itself. In other words: in each cycle through the `for` loop, the two elements are not themselves compared. What *is* compared is some derived, calculated value: `int(item)`, or `abs(item)`, or `item["gpa"]`.

It turns out we can abstract out that calculation, using what we'll call a *key function*. A key function is a function that takes exactly one argument—an element in the list. It returns the derived value used in the comparison. In fact, `int` works like a function,

even though it's technically a type, because `int("42")` returns 42.[3] So types and other callables work, as long as we can invoke it like a one-argument function.

This lets us define a very generic max function:

```
def max_by_key(items, key):
    biggest = items[0]
    for item in items[1:]:
        if key(item) > key(biggest):
            biggest = item
    return biggest
```

```
>>> # Old way:
... max_by_int_value(nums)
'30'
>>> # New way:
... max_by_key(nums, int)
'30'
>>> # Old way:
... max_by_abs(integers)
-20
>>> # New way:
... max_by_key(integers, abs)
-20
```

Pay attention: you are passing the function object itself—`int` and `abs`. You are *not* invoking the key function in any direct way. In other words, you write `int`, not `int()`. This function object is then called as needed by `max_by_key()`, to calculate the derived value:

```
        # key is actually int, abs, etc.
        if key(item) > key(biggest):
```

To sort the students by GPA, we need a function extracting the "`gpa`" key from each student dictionary. There is no built-in function that does this, but we can define our own and pass it in:

```
>>> # Old way:
... max_by_gpa(students)
{'gpa': 3.8, 'name': 'Jane Jones', 'major': 'chemistry'}

>>> # New way:
... def get_gpa(who):
...     return who["gpa"]
...
>>> max_by_key(students, get_gpa)
{'gpa': 3.8, 'name': 'Jane Jones', 'major': 'chemistry'}
```

3 Python uses the word *callable* to describe something that can be called like a function. This can be an actual function, a type or class name, or an object defining the `__call__` magic method. Key functions are frequently actual functions, but they can be any callable.

Again, notice `get_gpa` is a function object, and we are passing that function itself to `max_by_key`. We never invoke `get_gpa` directly; `max_by_key` does that automatically.

You may be realizing now just how powerful this can be. In Python, functions are simply objects—just as much as an integer, or a string, or an instance of a class is an object. You can store functions in variables, pass them as arguments to other functions, and even return them from other function and method calls. This all provides new ways for you to encapsulate and control the behavior of your code.

The Python standard library demonstrates some excellent ways to use such functional patterns. Let's look at a key (ha!) example.

Key Functions in Python

Earlier, you learned the built-in `max()` function doesn't magically do what we want when sorting a list of numbers-as-strings:

```
>>> nums = ["12", "7", "30", "14", "3"]
>>> max(nums)
'7'
```

Again, this isn't a bug. `max()` just compares elements according to the data type, and `"7" > "12"` evaluates to `True`. But it turns out `max()` is customizable. You can pass it a key function!

```
>>> max(nums, key=int)
'30'
```

The value of `key` is a function taking one argument—an element in the list—and returning a value for comparison. But `max()` isn't the only built-in that accepts a key function. `min()` and `sorted()` do as well:

```
>>> # Default behavior...
... min(nums)
'12'
>>> sorted(nums)
['12', '14', '3', '30', '7']
>>>
>>> # And with a key function:
... min(nums, key=int)
'3'
>>> sorted(nums, key=int)
['3', '7', '12', '14', '30']
```

Many algorithms can be cleanly expressed using `min()`, `max()`, or `sorted()`, along with an appropriate key function. Sometimes a built-in (like `int` or `abs`) will provide what you need, but often you'll want to create a custom function. Since this is so commonly needed, the `operator` module provides some helpers. Let's revisit the example of a list of student records.

```
>>> student_joe = {'gpa': 3.7, 'major': 'physics',
        'name': 'Joe Smith'}
>>> student_jane = {'gpa': 3.8, 'major': 'chemistry',
        'name': 'Jane Jones'}
>>> student_zoe = {'gpa': 3.4, 'major': 'literature',
        'name': 'Zoe Fox'}
>>> students = [student_joe, student_jane, student_zoe]
>>>
>>> def get_gpa(who):
...     return who["gpa"]
...
>>> sorted(students, key=get_gpa)
[{'gpa': 3.4, 'major': 'literature', 'name': 'Zoe Fox'},
 {'gpa': 3.7, 'major': 'physics', 'name': 'Joe Smith'},
 {'gpa': 3.8, 'major': 'chemistry', 'name': 'Jane Jones'}]
```

This is effective, and a fine way to solve the problem. Alternatively, the `operator` module's `itemgetter()` creates and returns a key function that looks up a named dictionary field:

```
>>> from operator import itemgetter
>>>
>>> # Sort by GPA...
... sorted(students, key=itemgetter("gpa"))
[{'gpa': 3.4, 'major': 'literature', 'name': 'Zoe Fox'},
 {'gpa': 3.7, 'major': 'physics', 'name': 'Joe Smith'},
 {'gpa': 3.8, 'major': 'chemistry', 'name': 'Jane Jones'}]
>>>
>>> # Now sort by major:
... sorted(students, key=itemgetter("major"))
[{'gpa': 3.8, 'major': 'chemistry', 'name': 'Jane Jones'},
 {'gpa': 3.4, 'major': 'literature', 'name': 'Zoe Fox'},
 {'gpa': 3.7, 'major': 'physics', 'name': 'Joe Smith'}]
```

Notice `itemgetter()` is a function that creates and returns a function—itself a good example of how to work with function objects. In other words, the following two key functions are completely equivalent:

```
# What we did above:
def get_gpa(who):
    return who["gpa"]

# Using itemgetter instead:
from operator import itemgetter
get_gpa = itemgetter("gpa")
```

This is how you use `itemgetter()` when the sequence elements are dictionaries. It also works when the elements are tuples or lists. Just pass a number index instead:

```
>>> # Same data, but as a list of tuples.
... student_rows = [
...     ("Joe Smith", "physics", 3.7),
...     ("Jane Jones", "chemistry", 3.8),
```

```
...      ("Zoe Fox", "literature", 3.4),
...      ]
>>>
>>> # GPA is the 3rd item in the tuple, i.e. index 2.
... # Highest GPA:
... max(student_rows, key=itemgetter(2))
('Jane Jones', 'chemistry', 3.8)
>>>
>>> # Sort by major:
... sorted(student_rows, key=itemgetter(1))
[('Jane Jones', 'chemistry', 3.8),
 ('Zoe Fox', 'literature', 3.4),
 ('Joe Smith', 'physics', 3.7)]
```

`operator` also provides `attrgetter()` for creating key functions based on an attribute of the element, and `methodcaller()` for creating key functions based off a method's return value—useful when the sequence elements are instances of your own class:

```
class Student:
    def __init__(self, name, major, gpa):
        self.name = name
        self.major = major
        self.gpa = gpa
    def __repr__(self):
        return f"{self.name}: {self.gpa}"

>>> student_objs = [
...      Student("Joe Smith", "physics", 3.7),
...      Student("Jane Jones", "chemistry", 3.8),
...      Student("Zoe Fox", "literature", 3.4),
...      ]
>>> from operator import attrgetter
>>> sorted(student_objs, key=attrgetter("gpa"))
[Zoe Fox: 3.4, Joe Smith: 3.7, Jane Jones: 3.8]
```

While these are sometimes directly useful, they are also valuable as examples of good key functions. They demonstrate how to design key functions for real use cases, so you can see when you benefit from creating your own.

Conclusion

Every coder knows the common ways of using functions. But most only scratch the surface of what you can do with them. Much power is unlocked when you start thinking of functions as just another object type. In addition to the useful patterns in this chapter, it is also the foundation of more powerful metaprogramming techniques. You will learn one of the most important ones in the next chapter.

CHAPTER 4
Decorators

Python supports a powerful tool called the *decorator*. Decorators let you add rich features to groups of functions and methods, without modifying them at all; untangle distinct, frustratingly intertwined concerns in your code, in ways not otherwise possible; and build powerful, extensible software frameworks. Many of the most popular and important Python libraries in the world leverage decorators. This chapter teaches you how to do the same.

A decorator is something you apply to a function or method. You've probably seen decorators before. There's a decorator called `property` often used in classes:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return self.first_name + " " + self.last_name

>>> person = Person("John", "Smith")
>>> print(person.full_name)
John Smith
```

Note that it is printing `person.full_name`, not `person.full_name()`.

For another example: in the Flask web framework, here is how you define a simple home page:

```
@app.route("/")
def hello():
    return "<html><body>Hello World!</body></html>"
```

The `app.route("/")` is a decorator, applied here to the function called `hello()`. So an HTTP GET request to the root URL ("/") will be handled by the `hello()` function.

A decorator works by *adding behavior around* a function[1]—meaning, lines of code which are executed before that function begins, after it returns, or both. It does not alter any lines of code *inside* the function. Typically, when you go to the trouble to define a decorator, you plan to use it on at least two different functions, usually more. Otherwise you'd just put the extra code inside the lone function, and not bother writing a decorator.

Using decorators is simple and easy; even someone new to programming can learn to use them quickly. Our objective in this chapter is different: to give you the ability to *write* your own decorators, in many different useful forms. This is not a beginner topic; it barely qualifies as intermediate. Writing decorators requires a deep understanding of several sophisticated Python features and how they play together. Most Python developers never learn how to create them. In this chapter, you will.

The Basic Decorator

Once a decorator is written, using it is easy. You just write `@` and the decorator name, on the line before you define a function:

```
@some_decorator
def some_function(arg):
    # blah blah
```

This applies the decorator called `some_decorator` to `some_function()`.[2] Now, it turns out this syntax with the `@` symbol is a shorthand. In essence, when byte-compiling your code, Python will translate the above into this:

```
def some_function(arg):
    # blah blah
some_function = some_decorator(some_function)
```

This is valid Python code too; it is what people did before the `@` syntax came along. The key here is the last line:

```
some_function = some_decorator(some_function)
```

First, understand that *a decorator is just a function*. That's it. It happens to be a function taking one argument, which is the function object being decorated. It then returns a different function. In the code snippet above you are defining a function,

1 Or method. When talking about the target of a decorator, "function" will always mean "function or method", unless I say otherwise.

2 For Java people: this looks just like Java annotations. However, it is *completely different*. Python decorators are not in any way similar.

initially called `some_function`. (Remember that `some_function` is a variable holding a function object, because that is what the `def` statement does.) That function object is passed to `some_decorator()`, which returns a *different* function object, which is finally stored in `some_function`.

To keep us sane, let's define some terminology:

- The *decorator* is what comes after the `@`. It's a function.
- The *bare function* is what is `def`'ed on the next line. It is, obviously, also a function.
- The end result is the *decorated function*. It's the final function that you actually call in your code.[3]

Your mastery of decorators will be most graceful if you remember one thing: a decorator is just a normal, boring function. It happens to be a function taking exactly one argument, which is itself a function. And when called, the decorator returns a *different* function.

Let's make this concrete. Here's a simple decorator which logs a message to stdout every time the decorated function is called.

```
def printlog(func):
    def wrapper(arg):
        print("CALLING: " + func.__name__)
        return func(arg)
    return wrapper

@printlog
def foo(x):
    print(x + 2)
```

This decorator creates a new function, called `wrapper()`, and returns that. This is then assigned to the variable `foo`, replacing the undecorated, bare function:

```
# Remember, this...
@printlog
def foo(x):
    print(x + 2)

# ...is the exact same as this:
def foo(x):
    print(x + 2)
foo = printlog(foo)
```

3 Some authors use the phrase "decorated function" to mean "the function that is decorated"—what I'm calling the "bare function". If you read a lot of blog posts, you'll find the phrase used both ways. (Sometimes in the same article.) This book will consistently use the definitions I give here.

Here's the result:

```
>>> foo(3)
CALLING: foo
5
```

At a high level, the body of `printlog()` does two things: define a function called `wrapper()`, then return it. Most decorators follow that structure. Notice `printlog()` does not modify the behavior of the original function `foo` itself; all `wrapper()` does is print a message to standard output, before calling the original (bare) function.

Once you've applied a decorator, the bare function isn't directly accessible anymore; you can't call it in your code. Its name now applies to the decorated version. But that decorated function internally retains a reference to the bare function, calling it inside `wrapper()`.

Generic Decorators

This version of `printlog()` has a shortcoming, though. Look what happens when I apply it to a different function:

```
>>> @printlog
... def baz(x, y):
...     return x ** y
...
>>> baz(3, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wrapper() takes 1 positional argument but 2 were given
```

Can you spot what went wrong?

`printlog()` is built to wrap a function taking exactly one argument. But `baz` has two, so when the decorated function is called, the whole thing blows up. There's no reason `printlog()` needs to have this restriction; all it's doing is printing the function name. You can fix it by declaring `wrapper()` with variable arguments:

```
# A MUCH BETTER printlog.
def printlog(func):
    def wrapper(*args, **kwargs):
        print("CALLING: " + func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

This decorator is compatible with *any* Python function:

```
>>> @printlog
... def foo(x):
...     print(x + 2)
...
>>> @printlog
```

```
... def baz(x, y):
...     return x ** y
...
>>> foo(7)
CALLING: foo
9
>>> baz(3, 2)
CALLING: baz
9
```

A decorator written this way, using variable arguments, will potentially work with functions and methods written *years* later—code the original developer never imagined. This structure has proven to be powerful and versatile.

```
# The prototypical form of Python decorators.
def prototype_decorator(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

We don't always do this, though. Sometimes you are writing a decorator that only applies to a function or method with a very specific kind of signature, and it would be an error to use it anywhere else. So feel free to break this rule when you have a reason.

Decorating Methods

Decorators apply to methods just as well as to functions. You often don't need to change anything: when the wrapper has a signature of `wrapper(*args, **kwargs)`, like `printlog()` does, it works just fine with any object's method. But sometimes you will see code like this:

```
# Not really necessary.
def printlog_for_method(func):
    def wrapper(self, *args, **kwargs):
        print("CALLING: " + func.__name__)
        return func(self, *args, **kwargs)
    return wrapper
```

This `wrapper()` has one required argument, named `self`. It works fine when applied to a method. But for the decorator I've written here, `self` is completely unnecessary, and in fact, it has a downside.

Simply defining `wrapper(*args, **kwargs)` causes `self` to be considered one of the `args`; such a decorator works just as well with both functions and methods. But if a wrapper is defined to require `self`, that means it must always be called with at least one argument. Suddenly you have a decorator that cannot be applied to functions which do not take at least one argument. (The fact that it's named `self` does not matter; it's just a temporary name for that first argument, inside the scope of `wrapper()`.)

You can apply this decorator to any method, and to some functions. But if you apply it to a function that takes *no* arguments, you'll get a runtime error.

Now, here's a different decorator:

```
# Using self makes sense in this case:
def enhanced_printlog_for_method(func):
    def wrapper(self, *args, **kwargs):
        print("CALLING: {} on object ID {}".format(
            func.__name__, id(self)))
        return func(self, *args, **kwargs)
    return wrapper
```

It could be applied like this:

```
class Invoice:
    def __init__(self, id_number, total):
        self.id_number = id_number
        self.total = total
        self.owed = total
    @enhanced_printlog_for_method
    def record_payment(self, amount):
        self.owed -= amount

inv = Invoice(42, 117.55)
print(f"ID of inv: {id(inv)}")
inv.record_payment(55.35)
```

Here's the output when you execute:

```
ID of inv: 4320786472
CALLING: record_payment on object ID 4320786472
```

This is a different story, because this `wrapper()`'s body explicitly uses the current object—a concept that only makes sense for methods. That makes the `self` argument perfectly appropriate. It prevents you from using this decorator on some functions. But since you intend this decorator to only be applied to methods, it would be an error to apply it to a function anyway.

When writing a decorator for methods, I recommend you get in the habit of making your wrapper only take `*args` and `**kwargs`, except when you have a clear reason to include `self`. After you've written decorators for a while, you'll be surprised at how often you end up using old decorators on new callables—both functions and methods —in ways you never imagined at first. A signature of `wrapper(*args, **kwargs)` preserves that flexibility. If the decorator turns out to need an explicit `self` argument, it's easy enough to put that in.

Data in Decorators

Some valuable decorator patterns rely on using variables inside the decorator function itself. This is *not* the same as using variables inside the *wrapper* function. Let me explain.

Imagine you need to keep a running average of what a chosen function returns. And further, you need to do this for a family of functions or methods. We can write a decorator called `running_average` to handle this. As you read, note carefully how `data` is defined and used:

```
def running_average(func):
    data = {"total" : 0, "count" : 0}
    def wrapper(*args, **kwargs):
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        print("Average of {} so far: {:.01f}".format(
             func.__name__, data["total"] / data["count"]))
        return val
    return wrapper
```

Each time the function is called, the average of all calls so far is printed out.[4] The decorator itself is called once for each bare function it is applied to. Then, each time the resulting decorated function is called in the code, the `wrapper()` function is what's actually executed.

So imagine applying `running_average` to a function like this:

```
@running_average
def foo(x):
    return x + 2
```

This executes `running_average()` as a function once, which creates an internal dictionary, named `data`, used to keep track of `foo`'s metrics.

But when you run `foo()`, you do NOT execute `running_average()` again. Instead, each time you run the decorated function `foo()`, it is calling the `wrapper()` which `running_average()` created internally. This means it can access `data` from its containing scope.

This may not all make sense yet, so let's step through it. Running `foo()` several times produces:

```
>>> foo(1)
Average of foo so far: 3.00
```

4 In a real application, you'd write the average to some kind of log sink, but we'll use `print()` here because it's convenient for learning.

```
3
>>> foo(10)
Average of foo so far: 7.50
12
>>> foo(1)
Average of foo so far: 6.00
3
>>> foo(1)
Average of foo so far: 5.25
3
```

The placement of `data` is important. Pop quiz:

- What happens if you move the line defining `data` up one line, outside the `running_average()` function?
- What happens if you move that line down, into the `wrapper()` function?

Looking at the code above, decide on your answers to these questions before reading further.

To answer the first question, here's what it looks like if you create `data` outside the decorator:

```
# This version has a bug.
data = {"total" : 0, "count" : 0}
def outside_data_running_average(func):
    def wrapper(*args, **kwargs):
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        print("Average of {} so far: {:.01f}".format(
             func.__name__, data["total"] / data["count"]))
        return func(*args, **kwargs)
    return wrapper
```

If you do this, *every* decorated function shares the exact same `data` dictionary! This actually doesn't matter if you only ever decorate just one function. But you never bother to write a decorator unless it's going to be applied to at least two:

```
@outside_data_running_average
def foo(x):
    return x + 2

@outside_data_running_average
def bar(x):
    return 3 * x
```

And that produces a problem:

```
>>> # First call to foo...
... foo(1)
Average of foo so far: 3.0
```

```
3
>>> # First call to bar...
... bar(10)
Average of bar so far: 16.5
30
>>> # Second foo should still average 3.00!
... foo(1)
Average of foo so far: 12.0
```

Because `outside_data_running_average()` uses the *same* `data` dictionary for all the functions it decorates, the statistics are conflated.

Now, for the other question: what if you define `data` inside `wrapper()`?

```
# This version has a DIFFERENT bug.
def running_average_data_in_wrapper(func):
    def wrapper(*args, **kwargs):
        data = {"total" : 0, "count" : 0}
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        print("Average of {} so far: {:.01f}".format(
             func.__name__, data["total"] / data["count"]))
        return func(*args, **kwargs)
    return wrapper

@running_average_data_in_wrapper
def foo(x):
    return x + 2
```

Look at the average as we call this decorated function multiple times:

```
>>> foo(1)
Average of foo so far: 3.0
3
>>> foo(5)
Average of foo so far: 7.0
7
>>> foo(20)
Average of foo so far: 22.0
22
```

Do you see why the running average is wrong? The `data` dictionary is reset *every time the decorated function is called*, because it is reset every time `wrapper()` is called. This is why it's important to consider the scope when implementing your decorator. Here's the correct version again (repeated so you don't have to skip back):

```
def running_average(func):
    data = {"total" : 0, "count" : 0}
    def wrapper(*args, **kwargs):
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
```

```
            print("Average of {} so far: {:.01f}".format(
                 func.__name__, data["total"] / data["count"]))
            return func(*args, **kwargs)
        return wrapper
```

So when exactly is `running_average()` executed? The decorator function itself is executed *exactly once* for *every* function it decorates. If you decorate N functions, `running_average()` is executed N times, so we get N different `data` dictionaries, each tied to one of the resulting decorated functions. This has nothing to do with how many times a decorated function is executed. The decorated function is, basically, one of the created `wrapper()` functions. That `wrapper()` can be executed many times, using the same `data` dictionary that was in scope when that `wrapper()` was defined.

This is why `running_average` produces the correct behavior:

```
@running_average
def foo(x):
    return x + 2

@running_average
def bar(x):
    return 3 * x
```

```
>>> # First call to foo...
... foo(1)
Average of foo so far: 3.0
3
>>> # First call to bar...
... bar(10)
Average of bar so far: 30.0
30
>>> # Second foo gives correct average this time!
... foo(1)
Average of foo so far: 3.0
3
```

Accessing Inner Data

What if you want to peek into `data`? The way we've written `running_average`, you can't—at least not with ordinary Python code. `data` persists because of the reference inside of `wrapper()`, and in a sense is veiled from outside access.

But there is an easy solution: simply assign `data` as an attribute to the `wrapper` object. For example:

```
# collectstats is much like running_average, but lets
# you access the data dictionary directly, instead
# of printing it out.
def collectstats(func):
    data = {"total" : 0, "count" : 0}
    def wrapper(*args, **kwargs):
```

```
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        return val
    wrapper.data = data
    return wrapper
```

See that line `wrapper.data = data`? Yes, you can do that. A function in Python is just an object, and in Python, you can add new attributes to objects by just assigning them. This conveniently annotates the decorated function:

```
@collectstats
def foo(x):
    return x + 2

>>> foo.data
{'total': 0, 'count': 0}
>>> foo(1)
3
>>> foo.data
{'total': 3, 'count': 1}
>>> foo(2)
4
>>> foo.data
{'total': 7, 'count': 2}
```

It's clear now why `collectstats` doesn't contain any print statement: you don't need one! We can check the accumulated numbers at any time, because this decorator annotates the function itself, with that `data` attribute.

Nonlocal Decorator State

Let's switch to another problem you might run into. Here's a decorator that counts how many times a function has been called:

```
# Watch out, this has a bug...
count = 0
def countcalls(func):
    def wrapper(*args, **kwargs):
        global count
        count += 1
        print(f"# of calls: {count}")
        return func(*args, **kwargs)
    return wrapper

@countcalls
def foo(x):
    return x + 2

@countcalls
def bar(x):
    return 3 * x
```

This version of `countcalls()` has a bug. Do you see it?

Here it is: this version stores `count` as global, meaning every function that is decorated will use the same variable:

```
>>> foo(1)
# of calls: 1
3
>>> foo(2)
# of calls: 2
4
>>> bar(3)
# of calls: 3
9
>>> bar(4)
# of calls: 4
12
>>> foo(5)
# of calls: 5
7
```

No good can come from this, so we need a better way. But the solution is trickier than it seems. Here's one attempt:

```
# Move count inside countcalls, and remove the
# "global count" line. But it still has a bug...
def countcalls(func):
    count = 0
    def wrapper(*args, **kwargs):
        count += 1
        print(f"# of calls: {count}")
        return func(*args, **kwargs)
    return wrapper
```

But that just creates a different problem:

```
>>> foo(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in wrapper
UnboundLocalError: local variable 'count' referenced before assignment
```

We can't use `global`, because it's not global. But we can use the `nonlocal` keyword:

```
# Final working version!
def countcalls(func):
    count = 0
    def wrapper(*args, **kwargs):
        nonlocal count
        count += 1
        print(f"# of calls: {count}")
        return func(*args, **kwargs)
    return wrapper
```

This finally works correctly:

```
>>> foo(1)
# of calls: 1
3
>>> foo(2)
# of calls: 2
4
>>> bar(3)
# of calls: 1
9
>>> bar(4)
# of calls: 2
12
>>> foo(5)
# of calls: 3
```

Applying `nonlocal` gives the `count` variable a special scope that is part way between local and global. Essentially, Python will search for the nearest enclosing scope that defines a variable named `count`, and use it like it's a global.

You may be wondering why we didn't need to use `nonlocal` with the first version of `running_average` above—here it is again, for reference:

```
def running_average(func):
    data = {"total" : 0, "count" : 0}
    def wrapper(*args, **kwargs):
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        print("Average of {} so far: {:.01f}".format(
             func.__name__, data["total"] / data["count"]))
        return func(*args, **kwargs)
    return wrapper
```

The line `count += 1` is actually modifying the value of the `count` variable itself, because it really means `count = count + 1`. And whenever you modify (instead of just read) a variable that was created in a larger scope, Python requires you to declare that's what you actually want, with `global` or `nonlocal`.

Here's the sneaky thing: when you write `data["count"] += 1`, *you are not actually modifying `data`!* Or rather, you're not modifying the *variable* named `data`, which points to a dictionary object. Instead, the statement `data["count"] += 1` invokes a *method* on the `data` object. (Specifically, `__setitem__()`.[5])

5 When you write `data["count"] += 1`, Python reads that as `data.__setitem__("count", data["count"] + 1)`.

This does change the state of the dictionary. But it doesn't make `data` point to a *different* dictionary. In contrast, `count += 1` makes `count` point to a different integer, so use `nonlocal` there.

Decorators That Take Arguments

Early in this chapter, I showed you an example decorator from the Flask framework:

```
@app.route("/")
def hello():
    return "<html><body>Hello World!</body></html>"
```

This is different from any decorator we've implemented so far, because it actually takes an argument. How do we write decorators that can do this? For example, imagine a family of decorators adding a number to the return value of a function:

```
def add2(func):
    def wrapper(n):
        return func(n) + 2
    return wrapper

def add4(func):
    def wrapper(n):
        return func(n) + 4
    return wrapper

@add2
def foo(x):
    return x ** 2

@add4
def bar(n):
    return n * 2
```

There is literally only one character difference between `add2` and `add4`. Wouldn't it be better if we can do something like this:

```
@add(2)
def foo(x):
    return x ** 2

@add(4)
def bar(n):
    return n * 2
```

We can. The key is to understand that `add` is actually *not* a decorator; it is a function that *returns* a decorator. In other words, `add` is a function that returns another function—since the returned decorator is, itself, a function.

To make this work, we write a function called `add()`, which creates and returns the decorator:

```
def add(increment):
    def decorator(func):
        def wrapper(n):
            return func(n) + increment
        return wrapper
    return decorator
```

It's easiest to understand from the inside out:

- The `wrapper()` function is just like in the other decorators. Ultimately, when you call `foo()` (the original function name), it's actually calling `wrapper()`.
- Moving up, we have the aptly named `decorator`. It is a function.
- At the top level is `add`. This is not a decorator. It's a function that returns a decorator.

Notice the closure here. The `increment` variable is encapsulated in the scope of the `add()` function. We cannot access its value outside the decorator, in the calling context. But we don't need to, because `wrapper()` itself has access to it.

Suppose the Python interpreter is parsing your program and encounters the following code:

```
@add(2)
def f(n):
    # ....
```

Python takes everything between the `@` symbol and the end-of-line character as a single Python expression—`add(2)` in this case. That expression is evaluated. This all happens *at compile time*. Evaluating the decorator expression means executing `add(2)`, which will return a function object. That function object is the decorator. It's named `decorator` inside the body of the `add()` function, but it doesn't really have a name at the top level; it's just applied to `f()`.

What can help you see more clearly is to think of functions as things that are stored in variables. In other words, if I write `def foo(x):` in my code, I can say to myself: "I'm creating a function called foo".

But there is another way to think about it. I could instead say: "I'm creating a function object, and storing it in a variable called foo". Believe it or not, this is closer to how Python actually works. So things like this are possible:

```
>>> def foo():
...     print("This is foo")
>>> baz = foo
>>> baz()
```

```
This is foo
>>> # foo and baz have the same id()... so they
... # refer to the same function object.
>>> id(foo)
4301663768
>>> id(baz)
4301663768
```

Now, back to `add`. As you realize `add(2)` returns a function object, it's easy to imagine storing that in a variable named `add2`. As a matter of fact, you can do this:

```
add2 = add(2)
@add2
def foo(x):
    return x ** 2
```

Remember that `@` is a shorthand:

```
# This...
@some_decorator
def some_function(arg):
    # blah blah

# ... is translated by Python into this:
def some_function(arg):
    # blah blah
some_function = some_decorator(some_function)
```

So for `add`, the following are all equivalent:

```
add2 = add(2) # Store the decorator in the add2 variable

# This function definition...
@add2
def foo(x):
    return x ** 2

# ... is translated by Python into this:
def foo(x):
    return x ** 2
foo = add2(foo)

# But also, this...
@add(2)
def foo(x):
    return x ** 2

# ... is translated by Python into this:
def foo(x):
    return x ** 2
foo = add(2)(foo)
```

Look over these variations, and trace through what's going on in your mind, until you understand how they are all equivalent. The expression `add(2)(foo)` in particular is interesting. Python parses this left-to-right. So it first executes `add(2)`, which returns a function object. In this expression, that function has no name; it is temporary and anonymous. The very next character is "`(`". This causes Python to take that anonymous function object, and immediately call it. It is called with the argument `foo` (which is the bare function—the function which we are decorating). The anonymous function then returns a *different* function object, which we finally store in the variable called `foo`, reassigning that variable name.

Study that previous paragraph until you fully get it. This is important.

Notice that in the line `foo = add(2)(foo)`, the name `foo` means something different each time it's used. When you write something like `n = n + 3`, the name `n` refers to something different on either side of the equals sign. In the exact same way, in the line `foo = add(2)(foo)`, the variable `foo` holds two different function objects on the left and right sides.

Class-Based Decorators

I lied to you.

I repeatedly told you that a decorator is just a function. Well, decorators are usually *implemented* as functions; that is true. But it is also possible to implement a decorator as a class. In fact, *any* decorator that you can implement as a function can be implemented with a class instead.

Why would you do this? For certain kinds of decorators, classes are better suited. They can be more readable, and otherwise easier to work with. In addition, they let you use the full feature set of Python's object system. For example, if you have a collection of related decorators, you can leverage inheritance to collect their shared code.

Most decorators are probably better implemented as functions, though it depends on whether you prefer object-oriented or functional abstractions. It is best to learn both ways, then decide which you prefer in your own code on a case-by-case basis.

Implementing Class-Based Decorators

The secret to decorating with classes is a magic method: `__call__()`. Any object can implement `__call__()` to make itself *callable*—meaning, it can be called like a function. Here's an example:

```
class Prefixer:
    def __init__(self, prefix):
        self.prefix = prefix
```

```
    def __call__(self, message):
        return self.prefix + message
```

We can call instances of this class like a function:

```
>>> simonsays = Prefixer("Simon says: ")
>>> simonsays("Get up and dance!")
'Simon says: Get up and dance!'
```

Just looking at `simonsays("Get up and dance!")` in isolation, you'd never guess it is anything other than a normal function. In fact, it's an instance of `Prefixer`.

What's happening is that any time you type an identifier name, followed by the "(" (left parenthesis) character, Python immediately translates that into the `__call__()` method. In fact, amazingly enough, this is true even for regular functions!

```
>>> def f(x):
...     return x + 1
>>> # This...
>>> f(2)
3
>>> # ... is the EXACT SAME as this:
>>> f.__call__(2)
3
```

When you use `__call__()`, you are hooking into something very fundamental in how Python works.

You can use `__call__()` to implement decorators. Before proceeding, think back to the `@printlog` decorator.Using this information about `__call__()`, how might you implement `printlog()` as a class instead of a function? I encourage you to pause and try implementing it yourself, before you proceed.

The basic approach is to pass `func` (the bare function) to the *constructor* of a decorator *class*, and adapt `wrapper()` to be the `__call__()` method. Like this:

```
class PrintLog:
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        print(f"CALLING: {self.func.__name__}")
        return self.func(*args, **kwargs)

# Compare to the function version you saw earlier:
def printlog(func):
    def wrapper(*args, **kwargs):
        print("CALLING: " + func.__name__)
        return func(*args, **kwargs)
    return wrapper

>>> @PrintLog
... def foo(x):
...     print(x + 2)
```

```
...
>>> @printlog
... def baz(x, y):
...     return x ** y
...
>>> foo(7)
CALLING: foo
9
>>> baz(3, 2)
CALLING: baz
9
```

From the user's point of view, `@Printlog` and `@printlog` work *exactly* the same.

Benefits of Class-Based Decorators

Class-based decorators have some advantages over function-based decorators. For one thing, the decorator is a class, which means you can leverage inheritance. So if you have a family of related decorators, you can reuse code between them. Here's an example:

```
import sys
class ResultAnnouncer:
    stream = sys.stdout
    prefix = "RESULT"
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        value = self.func(*args, **kwargs)
        self.stream.write(f"{self.prefix}: {value}\n")
        return value

class StdErrResultAnnouncer(ResultAnnouncer):
    stream = sys.stderr
    prefix = "ERROR"
```

Another benefit is when you prefer to accumulate state in object attributes, instead of a closure. For example, the `countcalls` function decorator (discussed in "Nonlocal Decorator State" on page 61) could be implemented as a class:

```
class CountCalls:
    def __init__(self, func):
        self.func = func
        self.count = 0
    def __call__(self, *args, **kwargs):
        self.count += 1
        print(f"# of calls: {self.count}")
        return self.func(*args, **kwargs)

@CountCalls
def foo(x):
    return x + 2
```

Notice this also lets us access `foo.count`, if we want to check the count outside of the decorated function. The function version didn't let us do this.[6]

When creating decorators which take arguments, the structure is a little different. In this case, the constructor accepts not the `func` object to be decorated, but the parameters on the decorator line. The `__call__()` method must take the `func` object, define a wrapper function, and return it—similar to simple function-based decorators:

```
# Class-based version of the "add" decorator above.
class Add:
    def __init__(self, increment):
        self.increment = increment
    def __call__(self, func):
        def wrapper(n):
            return func(n) + self.increment
        return wrapper
```

You then use it like you would any other argument-taking decorator:

```
>>> @Add(2)
... def foo(x):
...     return x ** 2
...
>>> @Add(4)
... def bar(n):
...     return n * 2
...
>>> foo(3)
11
>>> bar(77)
158
```

Any function-based decorator can be implemented as a class-based decorator; you simply adapt the decorator function itself to `__init__()`, and `wrapper()` to `__call__()`. It's possible to design class-based decorators which cannot be translated into a function-based form, though.

For certain complex decorators, some people feel that class-based decorators are easier to read than function-based ones. In particular, many people seem to find multi-nested `def` statements hard to reason about. Others (including your author) feel the opposite. This is a matter of preference,[7] and I recommend you practice with both styles before coming to your own conclusions.

---

6 This would also be a way to expose the `data` dict of the `@collectstats` decorator, from earlier in the chapter.

7 I believe it may have more to do with what language you "grew up" with. Someone who first learned to code in Java will tend to think best in terms of classes; someone with a mathematical background, or whose first coding language was more function oriented, will tend to think better in terms of nested functions.

In this section, I have capitalized the names of class-based decorators. I did this because I think it makes this section easier to understand. But that is not what I actually do in real code.

We have conflicting conventions here. Python class names are capitalized, and function names start with a lowercase letter. But whether a decorator is implemented as a function or class is an implementation detail. In fact, you may create a decorator as a function, and then later refactor it as a class—or vice versa. People using your decorator probably do not know or care whether it's a class or a function; to change its casing each time is a waste of effort, and breaks all the code using that decorator already.

So in my own code, I always use function-name conventions for decorators, even if they are implemented as a class. Essentially creating a higher-priority naming convention. I suggest you do this also; it will make it easier for everyone using the decorators you write, including you.

Decorators for Classes

I lied to you again. I said decorators are applied to functions and methods. Well, they can also be applied to classes.

Understand this has *nothing* to do with the last section's topic, on implementing decorators as classes. A decorator can be implemented as a function, or as a class; and that decorator can be applied to a function, or to a class. They are independent ideas. Here, we are talking about how to decorate classes instead of functions.

To introduce an example, let me explain Python's built-in `repr()` function. When called with one argument, this returns a string, meant to represent the passed object. It's similar to `str()`; the difference is that while `str()` returns a human-readable string, `repr()` is meant to return a string version of the Python code needed to recreate it. So imagine a simple `Penny` class:

```
class Penny:
    value = 1

penny = Penny()
```

Ideally, `repr(penny)` returns the string `"Penny()"`. But that's not what happens by default:

```
>>> class Penny:
...     value = 1
>>> penny = Penny()
>>> repr(penny)
'<__main__.Penny object at 0x10229ff60>'
```

You can fix this by implementing a `__repr__()` method on your classes, which `repr()` will use:

```
>>> class Penny:
...     value = 1
...     def __repr__(self):
...         return "Penny()"
>>> penny = Penny()
>>> repr(penny)
'Penny()'
```

Let's create a decorator that will automatically add a `__repr__()` method to any class. You might be able to guess how it works. Instead of a wrapper function, the decorator returns a class:

```
>>> def autorepr(cls):
...     def cls_repr(self):
...         return f"{cls.__name__}()"
...     cls.__repr__ = cls_repr
...     return cls
...
>>> @autorepr
... class Penny:
...     value = 1
...
>>> penny = Penny()
>>> repr(penny)
'Penny()'
```

It's suitable for classes with no-argument constructors, like `Penny`. Note how the decorator modifies `cls` directly. The original class is returned; that original class just now has a `__repr__()` method. Can you see how this is different from what we did with decorators of functions? With those, the decorator returned a new, different function object.

Another strategy for decorating classes is closer in spirit: creating a new subclass within the decorator, returning that subclass in place of the original:

```
def autorepr_subclass(cls):
    class NewClass(cls):
        def __repr__(self):
            return f"{cls.__name__}()"
    return NewClass
```

This has the disadvantage of creating a new type:

```
>>> @autorepr_subclass
... class Nickel:
...     value = 5
...
>>> nickel = Nickel()
>>> type(nickel)
<class '__main__.autorepr_subclass.<locals>.NewClass'>
```

The resulting object's type is not obviously related to the decorated class. That makes debugging harder, creates unclear log messages, and throws a greasy wrench into the middle of your inheritance hierarchy. For this reason, I recommend the first approach.

Class decorators tend to be less useful in practice than those for functions and methods. One real use is to automatically add dynamically generated methods. But they are more flexible than that. You can even express the singleton pattern using class decorators:

```
def singleton(cls):
    instances = {}
    def get_instance():
        if cls not in instances:
            instances[cls] = cls()
        return instances[cls]
    return get_instance

# There is only one Elvis.
@singleton
class Elvis:
    pass
```

Note the IDs are the same:

```
>>> elvis1 = Elvis()
>>> elvis2 = Elvis()
>>> id(elvis1)
4410742144
>>> id(elvis2)
4410742144
```

This `singleton` class decorator does not return a class; it returns a function object (`get_instance`). This means that in your code, `Elvis` will actually be that function object, rather than a class.

This is not without consequences; writing code like `isinstance(elvis1, Elvis)` will raise a `TypeError`, for example. This and other effects create code situations which are confusing, to say the least. It illustrates once again why we want to be careful about what is actually returned when we write decorators for classes.

Conclusion

Decorators are a powerful tool for metaprogramming in Python. It is no mistake that some of the most successful and famous Python libraries use them extensively in their codebase. When you learn to write decorators, not only do you add this power tool to your toolbox, but you also permanently level up your deep understanding of Python itself.

CHAPTER 5

Exceptions and Errors

Errors happen. That's why every practical programming language provides a rich framework for dealing with them.

Python's error model is based on *exceptions*. Some of you reading this are familiar with exceptions, and some are not. Some of you have used exceptions in other languages, but not yet with Python. This chapter is for all of you.

If you are familiar with how exceptions work in Java, C++, or C#, you'll find Python uses similar concepts, even if the syntax is rather different. And beyond those similarities lie uniquely Pythonic patterns.

We start with the basics. Even if you've used Python exceptions before, I recommend reading all of this chapter. Odds are you will learn useful things, even in sections which appear to discuss what you've seen before.

The Basic Idea

An *exception* is a way to interrupt the normal flow of code. When an exception occurs, the block of Python code will stop executing—literally in the middle of the line—and immediately jump to *another* block of code, designed to handle the situation.

Often an exception means an error of some sort, but it doesn't have to be. It can be used to signal anticipated events, which are best handled in an interrupt-driven way. Let's illustrate the common, simple cases first, before exploring more sophisticated patterns.

You've already encountered exceptions, even if you didn't realize it. Here's a little program using a `dict`:

```
# favdessert.py
def describe_favorite(category):
    "Describe my favorite food in a category."
    favorites = {
        "appetizer": "calamari",
        "vegetable": "broccoli",
        "beverage": "coffee",
    }
    return "My favorite {} is {}.".format(
        category, favorites[category])

message = describe_favorite("dessert")
print(message)
```

When run, this program exits with an error:

```
Traceback (most recent call last):
  File "favdessert.py", line 12, in <module>
    message = describe_favorite("dessert")
  File "favdessert.py", line 10, in describe_favorite
    category, favorites[category])
KeyError: 'dessert'
```

When you look up a missing dictionary key like this, we say that Python *raises* a `KeyError`. (In other languages, the terminology is "throw an exception". Same idea; Python uses the word "raise" instead of "throw".) That `KeyError` is an *exception*. In fact, most errors you see in Python are exceptions. This includes `IndexError` for bad indices (e.g., in lists), `TypeError` for incompatible types, `ValueError` for bad values, and so on. When an error occurs, Python responds by raising an exception.

Handling Exceptions

An exception needs to be handled. If not, your program will crash. You handle it with *try/except* blocks. They look like this:

```
# Replace the last few lines with the following:
try:
    message = describe_favorite("dessert")
    print(message)
except KeyError:
    print("I have no favorite dessert. I love them all!")
```

Notice the structure. You have the keyword `try`, followed by an indented block of code, immediately followed by `except KeyError`, which has its own block of code. We say the `except` block *catches* the `KeyError` exception.

Run the program with these new lines, and you get the following output:

```
I have no favorite dessert. I love them all!
```

Importantly, the new program exits successfully; its exit code to the operating system indicates "success" rather than "failure".

Here's how `try` and `except` work:

- Python starts executing lines of code in the `try` block.
- If Python gets to the end of the `try` block and no exceptions are raised, Python skips over the `except` block completely. None of its lines are executed, and Python proceeds to the next line after (if there is one).
- If an exception is raised anywhere in the `try` block, the program immediately stops—in the middle of the line, skipping the rest of that line and any remaining lines in the `try` block.
- Python then checks whether the exception type (`KeyError`, in this case) matches the `except` clause. If so, it jumps to the matching block's first line.
- If the exception does *not* match the `except` block, the exception ignores it, acting like the block isn't even there. If no higher-level code has an `except` block to catch that exception, the program will crash.

Let's wrap these lines of code in a function:

```
def print_description(category):
    try:
        message = describe_favorite(category)
        print(message)
    except KeyError:
        print(f"I have no favorite {category}. I love them all!")
```

Notice how `print_description()` behaves differently, depending on what you feed it:

```
>>> print_description("dessert")
I have no favorite dessert. I love them all!
>>> print_description("appetizer")
My favorite appetizer is calamari.
>>> print_description("beverage")
My favorite beverage is coffee.
>>> print_description("soup")
I have no favorite soup. I love them all!
```

Exceptions for Flow Control

Exceptions are not just for damage control. You will sometimes use them as a flow-control tool, to deal with ordinary variations you know can occur at runtime. Suppose, for example, your program loads data from a file, in JSON format. You import the `json.load()` function in your code:

```
from json import load
```

`json` is part of Python's standard library, so it's always available. Now, imagine there's an open-source library called `speedyjson`,[1] with a `load()` function just like what's in the standard library—except twice as fast. And your program works with *big* JSON files, so you want to preferentially use the `speedyjson` version when available. In Python, importing something that does not exist raises an `ImportError`:

```
# If speedyjson isn't installed...
>>> from speedyjson import load
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ImportError: No module named 'speedyjson'
```

How can you use `speedyjson` if it's there, yet gracefully fall back on `json` when it's not? Use a `try/except` block:

```
try:
    from speedyjson import load
except ImportError:
    from json import load
```

If `speedyjson` is installed and importable, `load()` will refer to its version of the function in your code. Otherwise you get `json.load()`.

A single `try` can have multiple `except` blocks. For example, `int()` will raise a `TypeError` if passed a nonsensical type; it raises `ValueError` if the type is acceptable, but its value cannot be converted to an integer:

```
try:
    value = int(user_input)
except ValueError:
    print("Bad value from user")
except TypeError:
    print("Invalid type (probably a bug)")
```

More realistically, you might log different error events[2] with different levels of severity:

```
try:
    value = int(user_input)
except ValueError:
    logging.error("Bad value from user: %r", user_input)
except TypeError:
    logging.critical(
        "Invalid type (probably a bug): %r", user_input)
```

1 Not a real library, so far as I know. But after this book is published, I'm sure one of you will make a library with that name, just to mess with me.

2 Especially in larger applications, exception handling often integrates with logging. See Chapter 9 for details.

If an exception is raised, Python will check whether its type matches the first `except` block. If not, it checks the next. The first matching `except` block is executed, and all others are skipped over entirely—so you will never have more than one of the `except` blocks executed for a given `try`. Of course, if none of them match, the exception continues rising until something catches it. (Or the process dies.)

There's a good rule of thumb that I suggest you start building as a habit now: *put as little code as possible in the `try` block*. You do this so your `except` block(s) will not catch or mask errors that they should not.

Finally Blocks

Sometimes you will want to have clean-up code that runs *no matter what*, even if an exception is raised. You can do this by adding a `finally` block:

```
try:
    line1
    line2
    # etc.
finally:
    line1
    line2
    # etc.
```

The code in the `finally` block is *always* executed. If an exception is raised in the `try` block, Python will immediately jump to the `finally` block, run its lines, then raise the exception. If an exception is not raised, Python will run all the lines in the `try` block, then run the lines in the `finally` block. It's a way to say, "Run these lines no matter what".

You can also have one (or more) `except` clauses:

```
try:
    line1
    line2
    # etc.
except FirstException:
    line1
    line2
    # etc.
except SecondException:
    line1
    line2
    # etc.
finally:
    line1
    line2
    # etc.
```

What's executed and when depends on whether an exception is raised. If not, the lines in the `try` block will run, followed by the lines in the `finally` block; none of the `except` blocks run. If an exception *is* raised, and it matches one of the `except` blocks, then the `finally` block runs *last*. The order is: the `try` block (up until the exception is raised), then the matching `except` block, and then the `finally` block.

What if an exception is raised, but there is no matching `except` block? The `except` blocks are ignored, because none of them match. The lines of code in `try` are executed, up until the exception is raised. Python immediately jumps to the `finally` block; when its lines finish, only then is the exception raised.

It's important to understand this ordering. When you include a `finally` block, and an exception is raised that does not match any `except` block, then the code in the `finally` block runs before that exception gets passed to the next higher level. A `finally` block is like insurance, for code that *must* run, no matter what.

Here's a good example. Imagine writing control code that does batch calculations on a fleet of cloud virtual machines. You issue an API call to rent them, and pay by the hour until you release them. Your code might look something like this:

```
# fleet_config is an object with the details of what
# virtual machines to start, and how to connect them.
fleet = CloudVMFleet(fleet_config)
# job_config details what kind of batch calculation to run.
job = BatchJob(job_config)
# .start() makes the API calls to rent the instances,
# blocking until they are ready to accept jobs.
fleet.start()
# Now submit the job. It returns a RunningJob handle.
running_job = fleet.submit_job(job)
# Wait for it to finish.
running_job.wait()
# And now release the fleet of VM instances, so we
# don't have to keep paying for them.
fleet.terminate()
```

Now imagine `running_job.wait()` raises a `socket.timeout` exception (which means the network connection has timed out). This causes a stack trace, and the program crashes; or maybe some higher-level code actually catches the exception.

Regardless, now `fleet.terminate()` is never called. Whoops. That could be *really* expensive.

To save your bank balance (or keep your job), rewrite the code using a `finally` block:

```
fleet = CloudVMFleet(fleet_config)
job = BatchJob(job_config)
try:
    fleet.start()
    running_job = fleet.submit_job(job)
```

```
        running_job.wait()
    finally:
        fleet.terminate()
```

This code expresses the idea: "no matter what, terminate the fleet of rented virtual machines." Even if an error in `fleet.submit_job(job)` or `running_job.wait()` makes the program crash, it will still call `fleet.terminate()` with its dying breath.

Dictionary Exceptions

Let's look at dictionaries again. When working directly with a dictionary, you can use the "if key in dictionary" pattern to avoid a `KeyError`, instead of `try/except` blocks:

```
# Another approach we could have taken with favdessert.py
def describe_favorite_or_default(category):
    'Describe my favorite food in a category.'
    favorites = {
        "appetizer": "calamari",
        "vegetable": "broccoli",
        "beverage": "coffee",
    }
    if category in favorites:
        message = "My favorite {} is {}.".format(
        category, favorites[category])
    else:
        message = f"I have no favorite {category}. I love them all!"
    return message

message = describe_favorite_or_default("dessert")
print(message)
```

The general pattern is:

```
# Using "if key in dictionary" idiom.
if key in mydict:
    value = mydict[key]
else:
    value = default_value

# Contrast with "try/except KeyError".
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

Many developers prefer using the "if key in dictionary" idiom, or using `dict.get()`. Sometimes you cannot do that, because the dict is buried in some code where you cannot get to it. A `try/except` block catching `KeyError` is your only choice then.

Exceptions Are Objects

An exception is an object: an instance of an exception class. `KeyError`, `IndexError`, `TypeError`, and `ValueError` are all built-in classes, which inherit from a base class called `Exception`. The code `except KeyError:` means "if the exception just raised is of type `KeyError`, run this block of code."

So far, I haven't shown you how to deal with those exception objects directly. And often, you won't need to. But sometimes you want more information about what happened, and capturing the exception object can help. Here's the structure:

```
try:
    do_something()
except ExceptionClass as exception_object:
    handle_exception(exception_object)
```

Here, *ExceptionClass* is some exception class, like `KeyError`, etc. In the `except` block, `exception_object` will be an instance of that class. You can choose any name for that variable; no one actually calls it `exception_object`. Most prefer shorter names like `ex`, `exc`, or `err`. The methods and contents of that object will depend on the kind of exception, but almost all will have an attribute called `args` that will be a tuple of what was passed to the exception's constructor. The `args` of a `KeyError`, for example, will have one element—the missing key:

```
# Atomic numbers of noble gases.
nobles = {'He': 2, 'Ne': 10,
  'Ar': 18, 'Kr': 36, 'Xe': 54}
def show_element_info(elements):
   for element in elements:
       print('Atomic number of {} is {}'.format(
             element, nobles[element]))
try:
    show_element_info(['Ne', 'Ar', 'Br'])
except KeyError as err:
    missing_element = err.args[0]
    print(f"Missing data for element: {missing_element}")
```

Running this code gives you the following output:

```
Atomic number of Ne is 10
Atomic number of Ar is 18
Missing data for element: Br
```

The interesting bit is in the `except` block. Writing `except KeyError as err` stores the exception object in the `err` variable. That lets us look up the offending key, by peeking in `err.args`. We could not get the offending key any other way, unless we want to modify `show_element_info()` (which we may not want to do, or perhaps *can't* do, as described before).

Let's walk through a more sophisticated example. In the `os` module, the `makedirs()` function will create a directory:

```
# Creates the directory "riddles", relative
# to the current directory.
import os
os.makedirs("riddles")
```

By default, if the directory already exists, `makedirs()` will raise `FileExistsError`. Imagine you are writing a web application, and need to create an upload directory for each new user. That directory should not exist yet; if it does, that's an error and needs to be logged. Our upload directory–creating function might look like this:

```
# First version....
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError:
        logging.error(
            "Upload dir for new user already exists")
```

It's great we are detecting and logging the error, but the error message isn't informative enough to be helpful. We at least need to know the offending username, but it's even better to know the directory's full path (so you don't have to dig in the code to remind yourself what `UPLOAD_ROOT` was set to).

Fortunately, `FileExistsError` objects have an attribute called `filename`. This is a string, and the path to the already-existing directory. We can use that to improve the log message:

```
# Better version!
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError as err:
        logging.error("Upload dir already exists: %s",
            err.filename)
```

Only the `except` block is different. That `filename` attribute is perfect for a useful log message.

Raising Exceptions

`ValueError` is a built-in exception that signals some data is of the correct type, but its format isn't valid. It shows up everywhere:

```
>>> int("not a number")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'not a number'
```

Your own code can raise exceptions, just like `int()` does. It should, in fact, so you have better error messages. (And sometimes for other reasons—more on that later.) You can do this with the *raise* statement. The most common form is this:

```
raise ExceptionClass(arguments)
```

For `ValueError` specifically, it might look like this:

```
def positive_int(value):
    number = int(value)
    if number <= 0:
        raise ValueError(f"Bad value: {value}")
    return number
```

Focus on the `raise` line in `positive_int()`. You simply create an instance of `ValueError`, and pass it directly to `raise`. Really, the syntax is `raise exception_object`—though usually you just create the object inline. `ValueError`'s constructor takes one argument, a descriptive string. This shows up in stack traces and log messages, so be sure to make it informative and useful:

```
>>> positive_int("-3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in positive_int
ValueError: Bad value: -3
>>> positive_int(-7.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in positive_int
ValueError: Bad value: -7.0
```

Let's show a more complex example. Imagine you have a `Money` class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    def __repr__(self):
        # Renders the object nicely on the prompt.
        return f"Money({self.dollars}, {self.cents})"
    # Plus other methods, which aren't important to us now.
```

Your code needs to create `Money` objects from string values, like "$140.75". The constructor takes dollars and cents, so you create a function to parse that string and instantiate `Money` for you:

```
import re
def money_from_string(amount):
    # amount is a string like "$140.75"
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

Your new function[3] works like this:

```
>>> money_from_string("$140.75")
Money(140, 75)
>>> money_from_string("$12.30")
Money(12, 30)
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in money_from_string
AttributeError: 'NoneType' object has no attribute 'group'
```

This error isn't clear; you must read the source and think about it to understand what went wrong. We have better things to do than decrypt stack traces. You can improve this function's usability by having it raise a `ValueError`.

```
import re
def money_from_string(amount):
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    # Adding the next two lines here
    if match is None:
        raise ValueError(f"Invalid amount: {amount}")
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

The error message is now much more informative:

```
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in money_from_string
ValueError: Invalid amount: 'Big money'
```

3 It is better to make this a class method of `Money`, rather than a separate function. That is a separate topic, though; see `@classmethod` in Chapter 6 for details.

Catching and Re-Raising

In an `except` block, you can re-raise the current exception. It's very simple; just write `raise` by itself, with no arguments:

```
try:
    do_something()
except ExceptionClass:
    handle_exception()
    raise
```

You don't need to store the exception object in a variable. It's a shorthand, exactly equivalent to this:

```
try:
    do_something()
except ExceptionClass as err:
    handle_exception()
    raise err
```

This "catch and release" only works in an `except` block. It requires some higher-level code to catch the exception and deal with it. But it enables several useful code patterns. One is when you want to delegate handling the exception to higher-level code, but also want to inject some extra behavior closer to the exception source. For example:

```
try:
    process_user_input(value)
except ValueError:
    logging.info("Invalid user input: %s", value)
    raise
```

If `process_user_input()` raises a `ValueError`, the `except` block will execute the logging line. Other than that, the exception propagates as normal.

This catch-and-release pattern is also useful when you need to execute code before deciding whether to re-raise the exception at all. Earlier, we used a `try/except` block pair to create an upload directory, logging an error if it already exists:

```
# Remember this? Python 3 code, from earlier.
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError as err:
        logging.error("Upload dir already exists: %s",
            err.filename)
```

This approach relies on `FileExistsError`, which was introduced in Python 3. How did people do this in Python 2? Even though you may never work with this older version, it's worth studying the different approach required, as it demonstrates a widely useful exception-handling pattern. Let's take a look.

`FileExistsError` subclasses the more general `OSError`. This exception type has been around since the early days of Python, and in Python 2, `makedirs()` simply raises `OSError`. But `OSError` can indicate many problems other than the directory already existing: a lack of filesystem permissions, a system call getting interrupted, or even a timeout over a network-mounted filesystem. We need a way to distinguish between these possibilities.

`OSError` objects have an `errno` attribute, indicating the precise error. These correspond to the variable `errno` in a C program, with different integer values meaning different error conditions. Most higher-level languages—including Python—reuse the constant names defined in the C API; in particular, the standard constant for "file already exists" is `EEXIST` (which happens to be set to the number 17 in most implementations). These constants are defined in the `errno` module in Python, so you just type `from errno import EEXIST` in your program.

In versions of Python with `FileExistsError`, the general pattern is:

- Optimistically create the directory.
- If `FileExistsError` is raised, catch it and log the event.

In Python 2, you must do this instead:

- Optimistically create the directory.
- If `OSError` is raised, catch it.
- Inspect the exception's `errno` attribute. If it's equal to `EEXIST`, this means the directory already existed; log that event.
- If `errno` is something else, it means we don't want to catch this exception here; re-raise the error.

The code:

```
# How to accomplish the same in Python 2.
import os
import logging
from errno import EEXIST
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
```

```
except OSError as err:
    if err.errno != EEXIST:
        raise
    logging.error("Upload dir already exists: %s",
        err.filename)
```

The only difference between the Python 2 and 3 versions is the `except` clause. But there's a lot going on there. First, we're catching `OSError` rather than `FileExistsError`. But we may or may not re-raise the exception, depending on the value of its `errno` attribute. Basically, a value of `EEXIST` means the directory already exists. So we log it and move on. Any other value indicates an error we aren't prepared to handle right here, so re-raise in order to pass it to higher-level code.

The Most Diabolical Python Antipattern

You know about *design patterns*: time-tested solutions to common code problems. And you've probably heard of *antipatterns*: solutions to code problems that *seem* to be good, but actually turn out to be harmful.

In Python, one antipattern is most harmful of all.

I wish I could avoid even telling you about it. If you don't know it exists, you can't use it in your code. Unfortunately, you might stumble on it somewhere and adopt it, not realizing the danger. So it is my duty to warn you.

Here's the punchline. The following is the most self-destructive code a Python developer can write:

```
try:
    do_something()
except:
    pass
```

Python lets you completely omit the argument to `except`. If you do that, it will catch *every exception*. That's pretty harmful right there; remember, the more pin-pointed your `except` clauses are, the more precise your error handling can be, without sweeping unrelated errors under the rug. And typing `except:` will sweep *every* unrelated error under the rug.

But it's much worse than that, because of the `pass` in the `except` clause. What `except: pass` does is silently and invisibly hide error conditions that you'd otherwise quickly detect and fix.

(Instead of "`except:`", you'll sometimes see variants like "`except Exception:`" or "`except Exception as ex:`". They amount to the same thing.)

This creates the *worst kind of bug*. Have you ever been troubleshooting a bug, and just couldn't figure out where in the codebase it came from, getting more and more frustrated as the hours roll by? *This is how you create that in Python.*

I first understood this antipattern after joining an engineering team, in an explosively growing Silicon Valley startup. We had a critical web service, that needed to be up 24/7. The engineers took turns being "on call" in case of a critical issue. An obscure Unicode bug somehow kept triggering, waking up engineers—in the middle of the night!—several times a week. But no one could figure out how to reproduce the bug, or even track down exactly how it was happening in the large code base.

After a few months of this nonsense, some of the senior engineers got fed up and devoted themselves to rooting out the problem. One senior engineer did nothing for *three full days* except investigate it, ignoring other responsibilities as they piled up. He made some progress, and took useful notes on what he found, but in the end, he did not figure it out. He ran out of time and had to give up.

Then, a second senior engineer took over. Using the first engineer's notes as a starting point, he also dug into it, ignoring emails and other commitments for another three full days. And he failed. He made progress, adding usefully to the notes. But in the end, he had to give up too.

Finally, after these six long days, they passed the torch to me—the new engineer on the team. I wasn't too familiar with the codebase, but their notes gave me a lot to go on. So I dove in on Day 7, and completely ignored everything else for six hours straight.

Finally, late in the day, I isolated the problem to a single block of code:

```
try:
    extract_address(location_data)
except:
    pass
```

That was it. The data in `location_data` was corrupted, causing the `extract_address()` call to raise a `UnicodeError`. Which the program then *completely silenced*. Not even producing a stack trace; simply moving on, as if nothing had happened.

After nearly *seven full days* of engineer effort, we pinpointed the error to this one block of code. I un-suppressed the exception, and almost immediately reproduced the bug—with a full stack trace.

Once I did that…can you guess how long it took us to fix the bug?

TEN MINUTES.

That's right. A full *week* of engineer time was wasted, all because this antipattern somehow snuck into our codebase. Had it not, then the first time it woke up an engineer, it would have been obvious what the problem was, and how to fix it. The code would have been patched by the end of the day, and we all would have immediately moved on.

The cruelty of this antipattern comes from how it completely hides *all* helpful information. Normally, when a bug causes a problem in your code, you can inspect the stack trace, identify what lines of code are involved, and start solving it. With The Most Diabolical Python Antipattern (TMDPA), none of that information is available. What line of code did the error come from? Which *file* in your Python application, for that matter? In fact, what was the exception type? Was it a `KeyError`? A `UnicodeError`? Or even a `NameError`, coming from a mistyped variable name? Was it `OSError`, and if so, what was its `errno`? You don't know. You *can't* know.

In fact, TMDPA often hides the fact that an error has even *occurred*. This is one of the ways bugs hide from you during development, then sneak into production, where they're free to cause real damage.

We never did figure out why the original developer wrote `except: pass` to begin with. I think that at the time, `location_data` may have sometimes been empty, causing `extract_address()` to innocuously raise a `ValueError`. In other words, if `ValueError` was raised, it was appropriate to ignore that and move on. By the time the other two engineers and I were involved, the codebase had changed so that was no longer how things worked. But the broad `except` block remained, like a land mine lurking in a lush field.

No one *wants* to wreak such havoc in their Python code, of course. People do this because they expect errors to occur in the normal course of operation, in some specific way. They are simply catching too broadly, without realizing the full implications.

So what should you do instead? There are two basic choices. In most cases, it's best to modify the `except` clause to catch a more specific exception. For the situation above, the following would have been a much better choice:

```
try:
    extract_address(location_data)
except ValueError:
    pass
```

Here, `ValueError` is caught and appropriately ignored. If `UnicodeError` raises, it propagates and (if not caught) the program crashes. That would have been *great* in our situation. The error log would have a full stack trace clearly telling us what happened, and we'd be able to fix it in 10 minutes.

As a variation, you may want to insert some logging:

```
try:
    extract_address(location_data)
except ValueError:
    logging.info(
        "Invalid location for user %s", username)
```

The other reason people write `except: pass` is a bit more valid. Sometimes, a code path simply must broadly catch all exceptions, and continue running regardless. This is common in the top-level loop for a long-running, persistent process. The problem is that `except: pass` hides all information about the problem, including that the problem even exists.

Fortunately, Python provides an easy way to capture that error event, and all the information you need to fix it. The `logging` module has a function called `exception()`, which will log your message *along with the full stack trace of the current exception*. So you can write code like this:

```
import logging

def get_number():
    return int('foo')
try:
    x = get_number()
except:
    logging.exception('Caught an error')
```

The log will contain the error message, followed by a formatted stack trace spread across several lines:

```
ERROR:root:Caught an error
Traceback (most recent call last):
  File "example-logging-exception.py", line 5, in <module>
    x = get_number()
  File "example-logging-exception.py", line 3, in get_number
    return int('foo')
ValueError: invalid literal for int() with base 10: 'foo'
```

This stack trace is *priceless*. Especially in more complex applications, it's often not enough to know the file and line number where an error occurs. It's at least as important to know *how* that function or method was called—what path of executed code led to it being invoked. Otherwise you can never determine what conditions lead to that function or method being called in the first place. The stack trace, in contrast, gives you everything you need to know.

Defeating This Diabolical Antipattern

I wish `except: pass` was not valid Python syntax. I think much grief would be spared if it was. But it's not my call, and changing it now is probably not practical.

Your only defense is to be vigilant. That includes educating your fellow developers. Does your team hold regular engineering meetings? Ask for five minutes at the next one to explain this antipattern, its cost to everyone's productivity, and the simple solutions.

Even better: if there are local Python or technical meetups in your area, volunteer to give a short talk. These meetups almost always need speakers, and you will be helping so many of your fellow developers.

There is a longer article explaining this situation at *https://powerfulpython.com/blog/the-most-diabolical-python-antipattern* . Simply sharing the URL will educate people too. And feel free to write your own blog post, with your own explanation of the situation, and how to fix it. Serve your fellow engineers by evangelizing this important knowledge.

Conclusion

Exceptions are a fundamental part of Python. But most Python coders only have a partial understanding of how they work. When you gain a deep understanding, one immediate benefit is that you decipher errors faster. Beyond that, you learn to write code that is more readable, more robust, and handles potential errors better. In short, your Python programs improve in every way.

CHAPTER 6

Classes and Objects: Beyond the Basics

This chapter assumes you are familiar with the basics of object-oriented programming (OOP) in Python: creating classes, defining methods, and simple inheritance. You will build on that knowledge in this chapter.

As with any object-oriented language, it's useful to learn about *design patterns*—reusable solutions to common problems involving classes and objects. A lot has been written about design patterns. And while much of it applies to Python, it tends to apply *differently*.

That is because many design-pattern books and articles are written for languages like Java, C++, and C#. But as a language, Python is different. Its dynamic typing, first-class functions, and other additions all mean the "standard" design patterns just work differently.

So let's learn what Pythonic OOP is *really* about.

Properties

Python objects have *attributes*. "Attribute" is a general term meaning "whatever is to the right of the dot" in an expression like `x.y` or `z.f()`. Member variables and methods are two kinds of attributes. But Python has another kind of attribute called *properties*.

A property is a hybrid: a cross between a method and a member variable. The idea is to create an attribute that acts like a member variable from the outside, but reading or writing to this attribute triggers method calls internally.

You'll set this up with a special decorator called `@property`. A simple example:

```
class Person:
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    @property
    def fullname(self):
        return self.firstname + " " + self.lastname
```

By instantiating this, you can access `fullname` like it is a member variable:

```
>>> joe = Person("Joe", "Smith")
>>> joe.fullname
'Joe Smith'
```

Look carefully for the actual member variables here. There are two, `firstname` and `lastname`, set in the constructor. This class also has a method called `fullname`. But after creating the instance, we reference `joe.fullname` as a member variable; we don't call `joe.fullname()` as a method. However, when you read the value of `joe.fullname`, the `fullname()` method is invoked.

This is all due to the `@property` decorator. When applied to a method, this decorator makes it inaccessible as a method. You must access it like a member variable. In fact, if you try to call it as a method, you get an error:

```
>>> joe.fullname()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

As defined above, `fullname` is read-only. We cannot modify it:

```
>>> joe.fullname = "Joseph Smith"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

In other words, Python properties are read-only by default. Another way of saying this is that `@property` automatically defines a *getter*, but not a *setter*.[1] If you want `fullname` to be writable, here is how you define its setter:

```
class Person:
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname
```

1 In OOP, when working with properties, we call the method which "reads" the current value the *getter*. And the method you call to set a new value is called the *setter*.

```
    @property
    def fullname(self):
        return self.firstname + " " + self.lastname

    @fullname.setter
    def fullname(self, value):
        self.firstname, self.lastname = value.split(" ", 1)
```

This lets you assign to `joe.fullname`:

```
>>> joe = Person("Joe", "Smith")
>>> joe.firstname
'Joe'
>>> joe.lastname
'Smith'
>>> joe.fullname = "Joseph Smith"
>>> joe.firstname
'Joseph'
>>> joe.lastname
'Smith'
```

So we have *two* methods named `fullname()`. The first one, decorated with `@property`, is dispatched (invoked) when you *read* the value of `joe.fullname`. The second one, decorated with `@fullname.setter`, is dispatched when you *assign* to `joe.fullname`. Python picks which to run, depending on whether you are getting or setting.

The first time I saw this, I had many questions. "Wait, why is this `fullname` method defined twice? And why is the second decorator named `@fullname`, and why does it have a `setter` attribute? How on earth does this even work?!"

The code is actually designed to work this way. The `@property` line, followed by `def fullname`, must come first. Those two lines create the property to begin with, and *also* create the getter. By "create the property", I mean that an object named `fullname` exists *in the namespace of the class*, and it has an attribute named `fullname.setter`. This `fullname.setter` is a decorator that is applied to the next `def fullname`, christening it as the setter for the `fullname` property.

It's okay to not fully understand how this works. A full explanation relies on understanding not only decorators, but also Python's descriptor protocol, which is beyond the scope of this chapter. Fortunately, you don't have to understand *how* it works in order to use it.

What you see here with the `Person` class is one way properties are useful: they are magic attributes which act like member variables, but their value is derived from other member variables. This denormalizes the object's data, and lets you access the attribute like it is a member variable. You'll see a situation where that is extremely useful later.

Property Patterns

Properties enable a useful collection of design patterns. One—as mentioned—is creating read-only member variables. In the first version of `Person`, the `fullname` "member variable" is a dynamic attribute; it does not exist on its own, but instead calculates its value at runtime.

It's also common to have the property backed by a single, non-public member variable. That pattern looks like this:

```
class Coupon:
    def __init__(self, amount):
        self._amount = amount
    @property
    def amount(self):
        return self._amount
```

This allows the class itself to modify the value internally, but prevent outside code from doing so:

```
>>> coupon = Coupon(1.25)
>>> coupon.amount
1.25
>>> coupon.amount = 1.50
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

In Python, prefixing a member variable or method by a single underscore signals it is protected; it should only be accessed internally, inside methods of that class or its subclasses.[2] This property pattern says, "You can read the value of this attribute, but you cannot change it".

Validation

There is another pattern between "regular member variable" and "read-only": the value can be changed, but you must validate it first. Suppose you and I are developing some software that manages live events. We write a `Ticket` class, representing tickets sold to attendees:

```
class Ticket:
    def __init__(self, price):
        self.price = price
    # And some other methods...
```

2 This is akin to `protected` in languages like Java. But unlike Java, Python does not enforce it. Instead, it is a convention you are expected to follow.

One day, we find a bug in our web UI that lets shifty customers adjust the price to a negative value. So we end up *paying* them to go to the concert. Not good!

The first priority is, of course, to fix the bug in the UI. But how do we modify our code to prevent this from ever happening again? Before reading further, look at the `Ticket` class and ponder—how could you use properties to make this kind of bug impossible in the future?

The answer: verify the new price is non-negative in the setter:

```
# Version 1...
class Ticket:
    def __init__(self, price):
        self._price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow non-negative prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

This lets the price be adjusted, but only to sensible values:

```
>>> t = Ticket(42)
>>> t.price = 24 # This is allowed.
>>> print(t.price)
24
>>> t.price = -1 # This is NOT.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in price
ValueError: Nice try
```

However, there's a defect in this new `Ticket` class. Can you spot it? (And how to fix it?)

The problem is that while we can't *change* the price to a negative value, this first version lets us *create* a ticket with a negative price to begin with. That's because we wrote `self._price = price` in the constructor. The solution is to use the *setter* in the constructor instead:

```
# Final version, with modified constructor. (Constructor
# is different; code for getter & setter is the same.)
class Ticket:
    def __init__(self, price):
        # instead of "self._price = price"
        self.price = price
    @property
    def price(self):
```

```
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

Yes, you can reference `self.price` in methods of the class. When we write `self.price = price`, Python translates this to calling the `price` setter—that is, the second `price()` method. This final version of `Ticket` centralizes all reads *and* writes of `self._price` in the property—a useful encapsulation technique. The idea is to centralize any special behavior for that member variable in the getter and setter, even for the class's internal code. In practice, sometimes other methods will need to violate this rule; if so, you simply reference `self._price` and move on. But as much as you can, only use the protected (underscore) member variable in the getter and setter, and that will naturally tend to boost the quality of your code.

Properties and Refactoring

Imagine writing a simple money class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    # And some other methods...
```

Suppose you put this class in a library many developers use: people on your current team, perhaps developers on different teams. Maybe you release it as open source, so developers around the world use and rely on this class.

One day you realize that many of `Money`'s methods—which do calculations on the money amount—could be simpler and more straightforward if they operated on the total number of cents, rather than dollars and cents separately. So you refactor the internal state:

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
```

This creates a *major* maintainability problem. Do you spot it?

Here's the trouble: your original `Money` has member variables named `dollars` and `cents`. And since many developers are using these variables, changing to `total_cents` breaks all their code!

```
money = Money(27, 12)
message = "I have {:d} dollars and {:d} cents."
# This line breaks, because there's no longer
# dollars or cents attributes.
print(message.format(money.dollars, money.cents))
```

If no one but you uses this class, there's no real problem—you can just refactor your own code. But otherwise, coordinating this change with everyone's different codebases is a nightmare. It becomes a barrier to improving your own code.

So, what do you do? Can you think of a way to handle this situation?

You get out of this mess using properties. You want two things to happen:

1. The class uses `total_cents` internally.
2. All code using `dollars` and `cents` continues to work, without modification.

You'll do this by replacing `dollars` and `cents` with `total_cents` internally, but also creating getters and setters for these attributes. Take a look:

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
    # Getter and setter for dollars...
    @property
    def dollars(self):
        # // is integer division
        return self.total_cents // 100
    @dollars.setter
    def dollars(self, new_dollars):
        self.total_cents = 100 * new_dollars + self.cents
    # And for cents.
    @property
    def cents(self):
        return self.total_cents % 100
    @cents.setter
    def cents(self, new_cents):
        self.total_cents = 100 * self.dollars + new_cents
```

Now, I can get and set `dollars` and `cents` all day:

```
>>> money = Money(27, 12)
>>> money.total_cents
2712
>>> money.cents
12
>>> money.dollars = 35
>>> money.total_cents
3512
```

Python's way of doing properties brings many benefits. In languages like Java, the following story can play out:

1. A newbie developer starts writing Java classes. They want to expose some state, so they create public member variables.
2. They use this class everywhere. Other developers use it too.
3. One day, the developer decides to change the name or type of that member variable, or even delete it entirely (like what we did with `Money`).
4. But that would break everyone's code. So they can't.

This is not a problem for Java developers in practice, because they quickly learn to make all their variables private by default—proactively creating getters and setters for *every* publicly exposed chunk of data. They realize this boilerplate is far less painful than the alternative, because if everyone must use the public getters and setters to begin with, you always have the freedom to make internal changes later.

This works well enough. But it *is* distracting, and just enough trouble that there's always the temptation to make that member variable public, and be done with it.

In Python, we have the best of both worlds. You can freely create member variables—which are public by default—and refactor them as properties if and when you ever need to. No one using your code even has to know.

The Factory Patterns

There are several design patterns with the word "factory" in their names. Their unifying idea is providing a handy, simplified way to create useful, potentially complex objects. The two most important forms are:

- Where the object's type is fixed, but we want to have several different ways to create it. This is called the *Simple Factory Pattern*.
- Where the factory dynamically chooses one of several different types. This is called the *Factory Method Pattern*.

Let's look at how you do these in Python.

Alternative Constructors: The Simple Factory

Imagine a simple `Money` class, suitable for currencies which have dollars and cents:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
```

We looked at this in the previous section, changing what member variables it has. But let's roll back, and focus instead on the constructor's interface. This constructor is convenient when we have the dollars and cents as separate integer variables. But there are many other ways to specify an amount of money. Perhaps you're modeling a giant jar of pennies:

```
# Emptying the penny jar...
total_pennies = 3274
dollars = total_pennies // 100
cents = total_pennies % 100
total_cash = Money(dollars, cents)
```

Suppose your code repeatedly splits pennies into dollars and cents, over and over. And you're tired of re-re-typing this calculation, plus there is a chance you could make a mistake eventually. You could change the constructor, but that means refactoring all `Money`-creating code, and perhaps a lot of code fits the current constructor better anyway. Some languages let you define several constructors, but Python makes you pick one.

In this case, you can usefully create a *factory function*. A factory function takes the data you *have*, uses that to calculate what the class constructor *needs*, then returns the instance. For example:

```
# Factory function taking a single argument, returning
# an appropriate Money instance.
def money_from_pennies(total_cents):
    dollars = total_cents // 100
    cents = total_cents % 100
    return Money(dollars, cents)
```

Imagine that, in the same codebase, you also need to parse strings like `"$140.75"`. Here's another factory function for that:

```
# Another factory, creating Money from a string amount.
import re
def money_from_string(amount):
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    if match is None:
        raise ValueError(f"Invalid amount: {amount}")
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

These are effectively alternate constructors: callables we can use with different arguments, which are used to create the final instance. But this approach has problems. First, it's awkward to have them as separate functions, defined outside of the class. But more importantly: what happens if you subclass `Money`? Suddenly `money_from_string()` and `money_from_pennies()` are worthless, because they are hard-coded to use `Money`.

Python solves these problems with a flexible and powerful feature: the `classmethod` decorator. Use it like this:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars = total_cents // 100
        cents = total_cents % 100
        return cls(dollars, cents)
```

The function `money_from_pennies()` is now a method of the `Money` class, called `from_pennies()`. But it has a new argument: `cls`. When applied to a method definition, `classmethod` modifies how that method is invoked and interpreted. The first argument is not `self`, which would be an *instance* of the class. The first argument is now *the class itself*. In the method body, `self` isn't mentioned at all; instead, `cls` is a variable holding the current class object—`Money` in this case. So the last line is creating a new instance of `Money`:

```
>>> piggie_bank_cash = Money.from_pennies(3217)
>>> type(piggie_bank_cash)
<class '__main__.Money'>
>>> piggie_bank_cash.dollars
32
>>> piggie_bank_cash.cents
17
```

Notice `from_pennies()` is invoked on the class itself, not an instance of the class. This is already nicer code organization. But now it works with inheritance:

```
>>> class TipMoney(Money):
...     pass
...
>>> tip = TipMoney.from_pennies(475)
>>> type(tip)
<class '__main__.TipMoney'>
```

This is the *real* benefit of class methods. You define it once on the base class, and all subclasses can leverage it, substituting their own type for `cls`. This makes class methods perfect for the simple factory in Python. The final line returns an instance of `cls`, using its regular constructor. And `cls` refers to whatever the current class is: `Money`, `TipMoney`, or some other subclass.

For the record, here's how to translate `money_from_string()`:

```
class Money:
    # ...
    def from_string(cls, amount):
        match = re.search(
```

```
            r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
        if match is None:
            raise ValueError(f"Invalid amount: {amount}")
        dollars = int(match.group('dollars'))
        cents = int(match.group('cents'))
        return cls(dollars, cents)
```

Class methods are a superior way to implement factories in Python. If we subclass `Money`, that subclass will have `from_pennies()` and `from_string()` methods that create objects of that subclass, without any extra work on our part. And if we change the name of the `Money` class, we only have to change it in one place, not three.

This form of the factory pattern is called *Simple Factory*, a name I don't love. I prefer to call it *Alternate Constructor*. Especially in the context of Python, it describes well what `@classmethod` is most useful for. And it suggests a general principle for designing your classes. Look at this complete code of the `Money` class, and I'll explain:

```
import re
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars = total_cents // 100
        cents = total_cents % 100
        return cls(dollars, cents)
    @classmethod
    def from_string(cls, amount):
        match = re.search(
            r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
        if match is None:
            raise ValueError(f"Invalid amount: {amount}")
        dollars = int(match.group('dollars'))
        cents = int(match.group('cents'))
        return cls(dollars, cents)
```

You can think of this class as having several constructors. As a general rule, you'll want to make `__init__()` the most generic one, and implement the others as class methods. Sometimes, that means one of the class methods will be used more often than `__init__()`.

When using a new class, most developers' intuition will be to reach for the default constructor first, without thinking to check the provided class methods—if they even know about that feature of Python in the first place. You may need to educate your teammates. (Hint: Good examples in the class's code docs go a long way.)

Dynamic Type: The Factory Method Pattern

This next factory pattern, called Factory Method, is quite different. The idea is that the factory will create an object, but will choose its type from one of several possibilities, dynamically deciding at runtime based on some criteria. It's typically used when you have one base class, and are creating an object that can be one of several different derived classes.

Let's see an example. Imagine you are implementing an image processing library, creating classes to read the image from storage. So you create a base `ImageReader` class, and several derived types:

```
import abc
class ImageReader(metaclass=abc.ABCMeta):
    def __init__(self, path):
        self.path = path
    @abc.abstractmethod
    def read(self):
        pass # Subclass must implement.
    def __repr__(self):
        return f"{self.__class__.__name__}({self.path})"

class GIFReader(ImageReader):
    def read(self):
        # Read a GIF

class JPEGReader(ImageReader):
    def read(self):
        # Read a JPEG

class PNGReader(ImageReader):
    def read(self):
        # Read a PNG
```

The `ImageReader` class is marked abstract, requiring subclasses to implement the `read()` method. So far, so good.

When reading an image file, if its extension is *.gif*, I want to use `GIFReader`. And if it is a JPEG image, I want to use `JPEGReader`, and so on. The logic is:

1. Analyze the file path name to get the extension.
2. Choose the correct reader class based on that.
3. Create the appropriate reader object.

This process is a prime candidate for automation. Let's define a little helper function:

```
def extension_of(path):
    # returns "png", "gif", "jpg", etc.
    position_of_last_dot = path.rfind('.')
    return path[position_of_last_dot+1:]
```

With these pieces, we can now define the factory:

```
def get_image_reader(path):
    image_type = extension_of(path)
    if image_type == 'gif':
        reader_class = GIFReader
    elif image_type == 'jpg':
        reader_class = JPEGReader
    elif image_type == 'png':
        reader_class = PNGReader
    else:
        raise ValueError(f"Unknown extension: {image_type}")
    return reader_class(path)
```

Classes in Python can be put in variables, just like any other object. We take full advantage here, by storing the appropriate `ImageReader` subclass in `reader_class`. Once we decide on the proper value, the last line creates and returns the reader object.

This correctly working code is already more concise, readable, and maintainable than what some languages force you to go through. But in Python, we can do even better. We can use the built-in dictionary type to make it more readable, and easier to update and maintain over time:

```
READERS = {
    'gif' : GIFReader,
    'jpg' : JPEGReader,
    'png' : PNGReader,
    }
def get_image_reader(path):
    reader_class = READERS[extension_of(path)]
    return reader_class(path)
```

Here we have a global variable mapping filename extensions to `ImageReader` subclasses. This lets us readably implement `get_image_reader()` in two lines. Finding the correct class is a simple dictionary lookup, and then we instantiate and return the instance. If we need to support new image formats in the future, we can simply add a line in the `READERS` definition. (And, of course, define its reader class.)

What if we encounter an extension not in the mapping, like *.tiff*? As written above, the code will raise a `KeyError`. That may be what we want. Or perhaps we want to catch that exception and re-raise a different exception. `ValueError` is a good choice; this is what the previous version of `get_image_reader()` raised.

Alternatively, we may want to fall back on some default. Let's create a new reader class, meant as an all-purpose fallback:

```
class RawByteReader(ImageReader):
    def read(self):
        # Read raw bytes
```

Then you can write the factory like:

```
def get_image_reader(path):
    try:
        reader_class = READERS[extension_of(path)]
    except KeyError:
        reader_class = RawByteReader
    return reader_class(path)
```

Or more briefly:

```
def get_image_reader(path):
    reader_class = READERS.get(extension_of(path), RawByteReader)
    return reader_class(path)
```

This design pattern is commonly called the *Factory Method* pattern, which wins my award for Worst Design Pattern Name in History. That name (which appears to originate from a Java implementation detail) fails to tell you anything about what this pattern is actually *for*. I myself call it the *Dynamic Type* pattern, which I feel is much more descriptive and useful.

So far, we have looked at patterns that are mostly confined to a single class. But there are richer patterns involving multiple codesigned classes, interacting with each other. Let's look at one.

The Observer Pattern

The Observer pattern provides a "one-to-many" relationship. That's a vague description, so let's make it more specific.

In the Observer pattern, there's one root object, called the *observable*. This object knows how to detect some kind of event of interest. It can literally be anything: a customer makes a new purchase; someone subscribes to an email list; or maybe it monitors a fleet of cloud instances, detecting when a machine's disk usage exceeds 75%. You use this pattern when the code to *reliably* detect the event of interest is at least slightly complicated; that detection code is encapsulated inside the observable.

In this pattern, you also have other objects, called *observers*, which need to know when that event occurs, so they can take some action in response. You don't want to reimplement the robust detection algorithm in each, of course. Instead, these observers register themselves with the observable. The observable then notifies each observer—by calling a method on that observer—for each event. This separation of concerns is the heart of the observer pattern.

I must tell you: I don't like the names of things in this pattern. The words "observable" and "observer" are a bit obscure, and sound confusingly similar—especially if your native tongue is not English. There is another terminology, however, which many developers find easier: *pub-sub*.

In this terminology, instead of an "observable," you create a *publisher* object, which watches for events. One or more *subscribers* (instead of "observers") ask that publisher to notify them when the event happens. I've found the pattern is easier to reason about when looked at in this way, so that is the terminology I'm going to use.[3]

Let's make this concrete, with code.

The Simple Observer

We'll start with the basic Observer pattern, as it's often documented in design pattern books—except we'll translate it to Python. In this simple form, each subscriber must implement a method called `update()`. Here's an example:

```
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print(f"{self.name} got message: {message}")
```

`update()` takes a string. It's okay to define an `update()` method taking other arguments, or even calling it something other than `update()`; the publisher and subscriber just need to agree on the protocol. But we'll use a single string argument.

Now, when a publisher detects an event, it notifies the subscriber by calling its `update()` method. Here's what a basic `Publisher` class looks like:

```
class Publisher:
    def __init__(self):
        self.subscribers = set()
    def register(self, who):
        self.subscribers.add(who)
    def unregister(self, who):
        self.subscribers.discard(who)
    def dispatch(self, message):
        for subscriber in self.subscribers:
            subscriber.update(message)
    # Plus other methods, for detecting the event.
```

Let's step through:

- A publisher needs to keep track of its subscribers, right? We'll store them in a set object, named `self.subscribers`, created in the constructor.
- A subscriber is added with `register()`. Its argument `who` is an instance of `Subscriber`. Who calls `register()`? It could be anyone. The subscriber can register itself, or some external code can register a subscriber with a specific publisher.

3 Technically, pub-sub is a more general architectural pattern that can apply to distributed systems. In contrast, the Observer pattern is always limited to what's inside a single process. That is the scope we will focus on here.

- `unregister()` is there in case a subscriber no longer needs to be notified of the events.
- When the event of interest occurs, the publisher notifies its subscribers by calling its `dispatch()` method. Usually this is invoked by the publisher itself, in some other method of the class (not shown) that implements the event-detection logic. It simply cycles through the subscribers, calling `update()` on each.

Using these two classes in code is straightforward enough:

```
# Create a publisher and some subscribers.
pub = Publisher()
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

# Register the subscribers, so they get notified.
pub.register(bob)
pub.register(alice)
pub.register(john)
```

Now, the publisher can dispatch messages:

```
# Send a message...
pub.dispatch("It's lunchtime!")
# John unsubscribes...
pub.unregister(john)
# ... and a new message is sent.
pub.dispatch("Time for dinner")
```

Here's the output from running the above:

```
John got message "It's lunchtime!"
Bob got message "It's lunchtime!"
Alice got message "It's lunchtime!"
Bob got message "Time for dinner"
Alice got message "Time for dinner"
```

This is the basic Observer pattern, and pretty close to how you'd implement the idea in languages like Java, C#, and C++. But Python's feature set differs from those languages. That means we can do different things.

A Pythonic Refinement

Python's functions are first-class objects. This means you can store a function in a variable—not the value returned when you call a function, but the function itself—as well as pass it as an argument to other functions and methods. Some other languages support this too (or something like it, such as function pointers), but Python's strong support gives us a convenient opportunity for this design pattern.

The standard Observer pattern requires the publisher to hard-code a certain method (usually named `update()`) that the subscriber must implement. But maybe you need to register a subscriber which doesn't have that method. What then? If it's your own class, you can just add it. If you are importing the subscriber class from another library (which you can't or don't want to modify), perhaps you can add the method by subclassing it.

Sometimes you can't do any of those things—or you *could*, but it's a lot of trouble, and you want to avoid it. What then?

Let's extend the traditional observer pattern, and make `register()` more flexible. Suppose you have these subscribers:

```
# This subscriber uses the standard "update"
class SubscriberOne:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print(f'{self.name} got message "{message}"')
# This one wants to use "receive"
class SubscriberTwo:
    def __init__(self, name):
        self.name = name
    def receive(self, message):
        print(f'{self.name} got message "{message}"')
```

`SubscriberOne` is the same subscriber class we saw before. `SubscriberTwo` is almost the same: instead of `update()`, it has a method named `receive()`. Let's modify `Publisher` so it can work with objects of either subscriber type:

```
class Publisher:
    def __init__(self):
        self.subscribers = dict()
    def register(self, who, callback=None):
        if callback is None:
            callback = who.update
        self.subscribers[who] = callback
    def dispatch(self, message):
        for callback in self.subscribers.values():
            callback(message)
    def unregister(self, who):
        del self.subscribers[who]
```

There's a lot going on here, so let's break it down. Look first at the constructor: it creates a `dict` instead of a set. You'll see why in a moment.

Now focus on `register()`:

```
    def register(self, who, callback=None):
        if callback is None:
            callback = who.update
        self.subscribers[who] = callback
```

It can be called with one or *two* arguments. With one argument, `who` is a subscriber of some sort, and `callback` defaults to `None`. In that case, the method body sets `callback` to `who.update`. Notice the lack of parentheses; `who.update` is a *method object*. It's just like a function object, except it happens to be tied to an instance. And just like a function object, you can store it in a variable, pass it as an argument to another function, and so on (refer to Chapter 3 for more details). So we're storing it in a variable called `callback`.

What if `register()` is called with two arguments? Here's how that might look:

```
pub = Publisher()
alice = SubscriberTwo('Alice')
pub.register(alice, alice.receive)
```

`alice.receive` is another method object; this object is assigned to `callback`. Regardless of whether `register()` is called with one argument or two, the last line inserts `callback` into the dictionary:

```
        self.subscribers[who] = callback
```

Take a moment to appreciate the remarkable flexibility of Python dictionaries. Here, you are using an arbitrary instance of either `SubscriberOne` or `SubscriberTwo` as a key. These two classes are unrelated by inheritance, so from Python's viewpoint they are completely distinct types. And for that key, you insert a *method object* as its value. Python does this seamlessly, without complaint! Many languages would make you jump through hoops to accomplish this.

Now it is clear why `self.subscribers` is a `dict` and not a `set`. Earlier, we only needed to keep track of who the subscribers were. Now, we also need to remember the callback for each subscriber. These are used in the `dispatch()` method:

```
    def dispatch(self, message):
        for callback in self.subscribers.values():
            callback(message)
```

`dispatch()` only needs to cycle through the values, because it just needs to call each subscriber's update method (even if it's not called `update()`). Notice we don't have to reference the subscriber object to call that method; the method object internally has a reference to its instance (i.e., its `"self"`), so `callback(message)` calls the right method on the right object. In fact, the only reason we keep track of subscribers at all is so we can `unregister()` them.

Let's put this together with a few subscribers:

```
pub = Publisher()
bob = SubscriberOne('Bob')
alice = SubscriberTwo('Alice')
john = SubscriberOne('John')

pub.register(bob, bob.update)
pub.register(alice, alice.receive)
pub.register(john)

pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

Here's the output:

```
Bob got message "It's lunchtime!"
Alice got message "It's lunchtime!"
John got message "It's lunchtime!"
Bob got message "Time for dinner"
Alice got message "Time for dinner"
```

Pop quiz. Look at the `Publisher` class again:

```
class Publisher:
    def __init__(self):
        self.subscribers = dict()
    def register(self, who, callback=None):
        if callback is None:
            callback = who.update
        self.subscribers[who] = callback
    def dispatch(self, message):
        for callback in self.subscribers.values():
            callback(message)
```

Does `callback` have to be a method of the subscriber? Or can it be a method of a different object, or something else? Think about this before you continue…

It turns out `callback` can be *any callable*, provided it has a signature compatible with how it's called in `dispatch()`. That means it can be a method of some other object, or even a normal function. This lets you register subscriber objects without an update method at all:

```
# This subscriber doesn't have ANY suitable method!
class SubscriberThree:
    def __init__(self, name):
        self.name = name
# ... but we can define a function...
todd = SubscriberThree('Todd')
def todd_callback(message):
    print(f'Todd got message "{message}"')
# ... and pass it to register:
```

```
pub.register(todd, todd_callback)
# And then, dispatch a message:
pub.dispatch("Breakfast is Ready")
```

Sure enough, this works:

```
Todd got message "Breakfast is Ready"
```

Several Channels

So far, we've assumed that the publisher watches for only one kind of event. But what if there are several? Can we create a publisher that knows how to detect all of them, and let subscribers decide which they want to know about?

To implement this, let's say a publisher has several *channels* that subscribers can subscribe to. Each channel notifies for a different event type. For example, if your program monitors a cluster of virtual machines, one channel signals when a certain machine's disk usage exceeds 75% (a warning sign, but not an immediate emergency); and another signals when disk usage goes over 90% (much more serious, and may begin to impact performance on that virtual machine). Some subscribers will want to know when the 75% threshold is crossed; some, the 90% threshold; and some might want to be alerted for both. What's a good way to express this in Python code?

Let's work with the mealtime-announcement code above. Rather than jumping right into the code, let's mock up the *interface* first. We want to create a publisher with two channels, like so:

```
# Two channels, named "lunch" and "dinner".
pub = Publisher(['lunch', 'dinner'])
```

This constructor is different; it takes a list of channel names. Let's also pass the channel name to `register()`, since each subscriber will register for one or more:

```
# Three subscribers, of the original type.
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

# Two args: channel name & subscriber
pub.register("lunch", bob)
pub.register("dinner", alice)
pub.register("lunch", john)
pub.register("dinner", john)
```

Now, on dispatch, the publisher needs to specify the event type. So just like with `register()`, we'll prepend a channel argument:

```
pub.dispatch("lunch", "It's lunchtime!")
pub.dispatch("dinner", "Dinner is served")
```

When correctly working, we'd expect this output:

```
Bob got message "It's lunchtime!"
John got message "It's lunchtime!"
Alice got message "Dinner is served"
John got message "Dinner is served"
```

Pop quiz (and if it's practical, pause here to write your own Python code): how would you implement this new, multi-channel `Publisher`?

There are several approaches, but the simplest I've found relies on creating a separate `subscribers` dictionary for each channel. Here's one approach:

```
class Publisher:
    def __init__(self, channels):
        # Create an empty subscribers dict
        # for every channel
        self.channels = { channel : dict()
                          for channel in channels }

    def register(self, channel, who, callback=None):
        if callback is None:
            callback = who.update
        subscribers = self.channels[channel]
        subscribers[who] = callback

    def dispatch(self, channel, message):
        subscribers = self.channels[channel]
        for callback in subscribers.values():
            callback(message)
```

This `Publisher` has a dict called `self.channels`, which maps channel names (strings) to subscriber dictionaries. `register()` and `dispatch()` are not too different: they simply have an extra step, in which `subscribers` is looked up in `self.channels`. I use that variable just for readability, and I think it's well worth the extra line of code:

```
# Works the same. But a bit less readable.
    def register(self, channel, who, callback=None):
        if callback is None:
            callback = who.update
        self.channels[channel][who] = callback
```

These are some variations of the general Observer pattern, and I'm sure you can imagine more. What I want you to notice are the options available in Python when you leverage function objects, and other Pythonic features.

Magic Methods

Suppose we want to create a class to work with angles, in degrees. We want this class to help us with some standard bookkeeping:

- An angle will be at least 0, but less than 360.
- If we create an angle outside this range, it automatically wraps around to an equivalent, in-range value:
  - If we add 270 degrees and 270 degrees, it evaluates to 180 degrees instead of 540 degrees.
  - If we subtract 180 degrees from 90 degrees, it evaluates to 270 degrees instead of -90 degrees.
  - If we multiply an angle by a real number, it wraps the final value into the correct range.
- And while we're at it, we want to enable all the other behaviors we normally want with numbers: comparisons like "less than" and "greater than or equal to" or "==" (i.e., equals); division (which doesn't normally require casting into a valid range, if you think about it); and so on.

Let's see how we might approach this, by creating a basic `Angle` class:

```
class Angle:
    def __init__(self, value):
        self.value = value % 360
```

The modular division in the constructor is kind of neat: if you reason through it with a few positive and negative values, you'll find the math works out correctly whether the angle is overshooting or undershooting. This meets one of our key criteria already: the angle is normalized to be from 0 up to 360.

But how does it handle addition? We get an error if we try it directly:

```
>>> Angle(30) + Angle(45)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Angle' and 'Angle'
>>>
```

We can easily define a method called `add`, which will let us write code like `angle3 = angle1.add(angle2)`. But it's better to reuse the familiar arithmetic operators everyone knows. Python lets us do that, through a collection of object hooks called *magic methods*. Magic methods let you define classes so that their instances can be used with all of Python's standard operators. That includes arithmetic (`+ - * / //`), equality (`==`), inequality (`!=`), comparisons (`< > >= \<=`), bit-shifting operations, and even concepts like exponentiation and absolute value.

Few classes will need all of these, but sometimes it's valuable to have them available. Let's see how they can improve our `Angle` type.

The pattern for each magic method is the same. For a given operation—say, addition—there is a special method name that starts with double-underscores. For addition, it's `__add__()`—the others also have sensible names. All you have to do is define that method, and you can use instances of your class with that operator.

When you discuss magic methods in face-to-face, verbal conversation, you'll find yourself saying things like "underscore underscore add underscore underscore" over and over. That's a lot of syllables. So people in the Python community use a kind of verbal abbreviation, with a word they invented: *dunder*. That's not a real word; Python people made it up. When you say "dunder foo", it means "underscore underscore foo underscore underscore". This isn't used in writing, because it's not needed—you can just write `__foo__`. But at Python gatherings, you'll sometimes hear people say it. Use it; it saves you a lot of energy when talking.

Anyway, back to dunder add—I mean, `__add__()`. For operations like addition—which take two values, and return a third—you write the method like this:

```
def __add__(self, other):
    return Angle(self.value + other.value)
```

The first argument needs to be called `self`, because this is Python. The second does not have to be called `other`, but often is. This lets us use the normal addition operator for arithmetic:

```
>>> total = Angle(30) + Angle(45)
>>> total.value
75
```

There are similar operators for subtraction, multiplication, and so on, as shown in Table 6-1:

Table 6-1. Arithmetic magic methods

Method	Operation
`__add__()`	`a + b`
`__sub__()`	`a - b`
`__mul__()`	`a * b`
`__truediv__()`	`a / b` (floating-point division)
`__mod__()`	`a % b`
`__pow__()`	`a ** b`

Essentially, Python translates `a + b` to `a.__add__(b)`; `a % b` to `a.__mod__(b)`; and so on. You can also hook into bit-operation operators (see Table 6-2).

Table 6-2. Bit-operation magic methods

Method	Operation
`__lshift__()`	`a << b`
`__rshift__()`	`a >> b`
`__and__()`	`a & b`
`__xor__()`	`a ^ b`
`__or__()`	`a | b`

So `a & b` translates to `a.__and__(b)`, for example.

Since `__and__()` corresponds to the bitwise-and operator (for expressions like "`foo & bar`"), you might wonder what the magic method is for *logical*-and ("`foo and bar`"), or logical-or ("`foo or bar`"). Sadly, there is none; because of how Python's Boolean logic short-circuits, there is not really a good way to do magic methods for them. For this reason, sometimes libraries will hijack the `&` and `|` operators to mean logical and/or, instead of bitwise and/or.

The default representation of an `Angle` object isn't very useful:

```
>>> Angle(30)
<__main__.Angle object at 0x106df9198>
```

It tells us the type, and the hex object ID, but we'd rather it tell us something about the value of the angle. There are two magic methods that can help. The first is `__str__()`, which is used when printing a result:

```
class Angle:
    # ...
    def __str__(self):
        return f"{self.value} degrees"
```

The `print()` function uses this, and so do `str()` and the string formatting operations:

```
>>> print(Angle(30))
30 degrees
>>> print(f"{Angle(30) + Angle(45)}")
75 degrees
>>> print("{}".format(Angle(30) + Angle(45)))
75 degrees
>>> str(Angle(135))
'135 degrees'
>>> some_angle = Angle(45)
>>> f"{some_angle}"
'45 degrees'
```

Sometimes you want a string representation that is more precise, which might be at odds with a human-friendly representation. Imagine you have several subclasses (for

instance, `PitchAngle` and `YawAngle` in some kind of aircraft-related library), and you want an easy way to log the exact type and arguments needed to recreate the object. Python provides a second magic method for this purpose, called `__repr__()`:

```
class Angle:
    # ...
    def __repr__(self):
        return f"Angle({self.value})"
```

You access this by calling either the `repr()` built-in function, or by passing the `!r` conversion to the formatting string:

```
>>> repr(Angle(75))
'Angle(75)'
>>> print('{!r}'.format(Angle(30) + Angle(45)))
Angle(75)
>>> print(f"{Angle(30) + Angle(45)!r}")
Angle(75)
```

You can think of both of these as working like `str()`, but invoking `__repr__()` instead of `__str__()`.

The official guideline is that the output of `__repr__()` can be passed to `eval()` to recreate the object exactly. It's not enforced by the language, and is not always practical, or even possible. But when you can follow that guideline, it is useful for logging and debugging.

We also want to be able to compare two `Angle` objects. The most basic comparison is equality, provided by `__eq__()`. It should return `True` or `False`:

```
class Angle:
    # ...
    def __eq__(self, other):
        return self.value == other.value
```

If defined, this method is used by the == operator:

```
>>> Angle(3) == Angle(3)
True
>>> Angle(7) == Angle(1)
False
```

By default, the == operator is based on the object ID. So an expression like `x == y` evaluates to `True` if `x` and `y` have the same ID, and otherwise evaluates to `False`. That is rarely useful:

```
>>> class BadAngle:
...     def __init__(self, value):
...         self.value = value
...
>>> BadAngle(3) == BadAngle(3)
False
```

What's left are the fuzzier comparison operations: less than, greater than, and so on. Python's documentation calls these "rich comparison" methods, so you can feel wealthy when using them (see Table 6-3).

Table 6-3. Rich comparison magic methods

Method	Operation
`__lt__()`	less than (<)
`__le__()`	less than or equal (\<=)
`__gt__()`	greater than (>)
`__ge__()`	greater than or equal (>=)

For example:

```
class Angle:
    # ...
    def __gt__(self, other):
        return self.value > other.value
```

Now the greater-than operator works correctly:

```
>>> Angle(100) > Angle(50)
True
```

Similarly, with `__ge__()`, `__lt__()`, etc. If you don't define these, you get an error:

```
>>> BadAngle(8) > BadAngle(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: BadAngle() > BadAngle()
```

`__gt__()` and `__lt__()` are reflections of each other. What that means is that, in many cases, you only have to define one of them. Suppose you implement `__gt__()` but not `__lt__()`, then do this:

```
>>> a1 = Angle(3)
>>> a2 = Angle(7)
>>> a1 < a2
True
```

This works thanks to some just-in-time introspection the Python runtime does. The `a1 < a2` is translated to `a1.__lt__(a2)`. If `Angle.__lt__()` is indeed defined, that method is executed, and the expression evaluates to its return value.

`a1 < a2` is true if and only if `a2 > a1`. For this reason, if `__lt__()` does not exist, but `__gt__()` does, then Python will rewrite the angle comparison: `a1.__lt__(a2)` becomes `a2.__gt__(a1)`. This is then evaluated, and the expression `a1 < a2` is set to its return value. There are some situations where you will need to define both, for example, if the comparison is based on several member variables.

Rebelliously Misusing Magic Methods

Magic methods are interesting enough, and quite handy when you need them. But depending on the kind of applications you work on, you will rarely need to define a class whose instances can be added, subtracted, or compared.

Things get much more interesting, though, when you don't follow the rules.

Here's a fascinating fact: methods like `__add__()` are *supposed* to do addition. But it turns out that Python does not enforce this. And methods like `__gt__()` are *supposed* to return `True` or `False`. But if you write a `__gt__()` which returns something that isn't a `bool`…Python will not complain at all.

This creates *amazing* possibilities.

To illustrate, let me tell you about Pandas. You probably know that Pandas is an excellent data-processing library. It's become extremely popular among data scientists who use Python. Pandas has a convenient data type called a `DataFrame`. It represents a two-dimensional collection of data, organized into rows, with labeled columns:

```
import pandas
df = pandas.DataFrame({
        'A': [-137, 22, -3, 4, 5],
        'B': [10, 11, 121, 13, 14],
        'C': [3, 6, 91, 12, 15],
    })
```

There are several ways to create a `DataFrame`; here I've chosen to use a dictionary.[4] The keys are column names; the values are lists, which become that column's data. So you visually rotate each list 90 degrees:

```
>>> print(df)
     A    B   C
0 -137   10   3
1   22   11   6
2   -3  121  91
3    4   13  12
4    5   14  15
```

The rows are numbered for you, and the columns nicely labeled in a header. The `A` column, for example, has different positive and negative numbers.

Now, one of the many useful things you can do with a `DataFrame` is filter out rows meeting certain criteria. This doesn't change the original DataFrame; instead, it

4 Which you will rarely do in real code (you will ingest from a CSV file or something instead), but it is convenient for demonstrating here.

creates a *new* DataFrame, containing just the rows you want. For example, you can say, "Give me the rows of `df` in which the `A` column has a positive value":

```
>>> positive_a = df[df.A > 0]
>>> print(positive_a)
    A   B   C
1  22  11   6
3   4  13  12
4   5  14  15
```

All you have to do is pass in "`df.A > 0`" in the square brackets.

But there's something weird going on here. Take a look at the line in which `positive_a` is defined. Do you notice anything unusual there? Anything strange?

Here's what is odd: the expression "`df.A > 0`" ought to evaluate to either `True`, or `False`. Right? It's supposed to be a Boolean value with exactly *one bit* of information. But the source dataframe, `df`, has many rows. Real dataframes can easily have tens of thousands, even *millions* of rows of data. There's no way a Boolean literal can express which of those rows to keep and which to discard. How does this even work?

Turns out, it's not Boolean at all:

```
>>> comparison = (df.A > 0)
>>> type(comparison)
<class 'pandas.core.series.Series'>
>>> print(comparison)
0    False
1     True
2    False
3     True
4     True
Name: A, dtype: bool
```

Yes, you can do that, thanks to Python's dynamic type system. Python translates "`df.A > 0`" into "`df.A.__gt__(0)`". And that `__gt__()` method doesn't have to return a `bool`. In fact, in Pandas, it returns a `Series` object (which is like a vector of data), containing `True` or `False` for each row. And when that's passed into `df[]`—the square brackets being handled by the `__getitem__()` method—that `Series` object is used to filter rows.

To see what this looks like, let's re-invent part of the interface of Pandas. I'll create a library called `fakepandas`, which instead of `DataFrame` has a type called `Dataset`:

```
class Dataset:
    def __init__(self, data):
        self.data = data
        self.labels = sorted(data.keys())
    def __getattr__(self, label: str):
        # Makes references like df.A work.
```

```
            return Column(label)
        # Plus some other methods.
```

If I have a `Dataset` object named `ds`, with a column named `A`, the `__getattr__()` method causes references like `ds.A` to return a `Column` object:

```
import operator
class Column:
    def __init__(self, name):
        self.name = name
    def __gt__(self, value):
         return Comparison(self.name, value, operator.gt)
```

This `Column` class has a `__gt__()` method, which makes expressions like "`ds.A > 0`" return an instance of a class called `Comparison`. It represents a lazy calculation for when the actual filtering happens later. Notice its constructor arguments: a column name, a threshold value, and a callable to implement the comparison. (The `operator` module has a function called `gt()` that takes two arguments, expressing a greater-than comparison).

You can even support complex filtering criteria like `ds[ds.C + 2 < ds.B]`. It's all possible by leveraging magic methods in these unorthodox ways. If you care about the details, I wrote an article delving into that. My goal here isn't to tell you how to reinvent the Pandas interface, so much as to get you to realize what's possible.

Have you ever implemented a compiler? If so, you know the parsing phase is a significant development challenge. Using Python magic methods in this manner does much of the hard work of lexing and parsing for you. And the best part is how natural and intuitive the result can be for end users. You are essentially implementing a domain-specific language on top of regular Python syntax, but consistently enough that people quickly become fluent and productive within its rules. They often won't even think to ask why the rules seem to be bent; they won't notice that "`df.A > 0`" isn't acting like a Boolean. That's a clear sign of success. It means you have designed your library so well, other developers become effortlessly productive.

Conclusion

Most Python users know how to write simple classes and methods. But as you can see, Python's object system has a lot more to it than that. Learning more advanced OOP opens up great opportunities for you, and allows you to create code structures you would otherwise never produce. Take everything you learned in this chapter, and put it into action.

CHAPTER 7
Automated Testing

Writing automated tests is one of those things that separates average developers from the best in the world. Master this skill, and you will be able to write *far* more complex and powerful software than you ever could before. It is a superpower that changes the arc of your career.

Some of you have, so far, little or no experience writing automated tests, in any language. This chapter is primarily written for you. It introduces many fundamental ideas of test automation, explains the problems it is supposed to solve, and teaches how to apply Python's tools for doing so.

Some of you will have extensive experience using standard test frameworks in other languages (such as JUnit in Java, PHPUnit in PHP, and so on). Generally speaking, if you have mastered an xUnit framework in another language, and are fluent in Python, you may be able to start skimming Python's `unittest` module docs[1] and be productive in minutes. Python's test library, `unittest`, maps closely to how most xUnit libraries work.[2]

If you are more experienced, I believe it is worth your time to at least skim this chapter, and perhaps study it thoroughly. I have woven in useful, real-world wisdom for software testing in general, and for Python specifically. This includes topics like how to organize Python test code, writing *maintainable* test code, useful features like subtests, and even cognitive aspects of programming… like getting into an enjoyable, highly productive "flow" state via test-driven development.

1 *https://docs.python.org/3/library/unittest.html*

2 You may be in a third category, having a lot of experience with a non-xUnit testing framework. If so, you should probably pretend you're in the first group. You'll be able to move quickly.

With that in mind, let's start with the core ideas for writing automated tests. We'll then apply those ideas specifically to Python.

What Is Test-Driven Development?

An *automated test* is a program that tests another program. Generally, it tests a specific portion of that program: a function, a method, a class, or some group of these things chunked together. We call that portion the *system under test*, sometimes abbreviated as SUT.

If the system under test is working correctly, the automated test *passes*; if not, that test *fails*—which means it catches the error and immediately tells you what is wrong. Real applications accumulate many of these tests as development proceeds.

People have different names for different kinds of automated tests: unit tests, integration tests, end-to-end tests, etc. These distinctions are useful, but we won't need to worry about them right now. They all share the same foundation.

In this chapter, we do *test-driven development*, or TDD. Test-driven development means you start working on each new feature or bugfix by writing the automated test for it *first*. You write the test; run it, to verify that it fails (and make sure that test code is actually working); and only then write the actual code for the feature. You know it works when the test passes.

This is a different process from implementing the feature first, *then* writing a test for it. Writing the test first forces you to think through the interfaces of your code, answering the question, "How will I know my code is working?" That immediate benefit is useful, but it is not the whole story.

TDD's greatest midterm benefits are mostly cognitive. As you become competent and comfortable with its tactics and techniques, you learn to easily get into a state of flow—where you find yourself repeatedly implementing feature after feature, keeping your focus with ease for long periods of time. You can honestly surprise and delight yourself with how much you accomplish in a short period of time.

But even greater benefits emerge over time. We've all done substantial refactorings of a large codebase, changing fundamental aspects of its architecture.[3] Such refactorings—which threaten to break the application in confusing, hidden ways—become straightforward and safe when you have test code in place already, and use TDD to refactor from that foundation. You first update the tests: modifying where needed, and writing new tests as appropriate. Then all you have to do is make them pass. It

3 If you haven't done one of these yet, you will someday.

might still be a lot of work. But you can be confident in the correctness of your code, once the new tests pass.

Among developers who know how to write tests, some love to do TDD in their day-to-day work. Some like to do it part of the time; some hate it, and do it rarely or never. However, the absolute best way to quickly master unit testing is to strictly do TDD for a while. So I'll teach you how to do that. You do not have to do it forever if you don't want to.

Python's standard library ships with two modules for creating unit tests: `doctest` and `unittest`. Most engineering teams prefer `unittest`, as it is more full-featured than `doctest`. This isn't just a convenience. There is a low ceiling of complexity that `doctest` can handle, and real applications will quickly bump up against that limit. With `unittest`, the sky is more or less the limit.

And because `unittest` maps closely to the xUnit libraries used in many other languages, if you are already familiar with Python, and have used an xUnit library in any language, you will feel right at home with `unittest`. That said, `unittest` has some unique tools and idioms—partly because of differences in the Python language, and partly from unique extensions and improvements. We will learn the best of what `unittest` has to offer as we go along.

Another popular option is `pytest`. This is not in the standard library, but it is widely used. For brevity, we will focus on `unittest` in this chapter. Once you learn the principles, picking up `pytest` is straightforward.

Unit Tests and Simple Assertions

Imagine a class representing an angle:

```
>>> small_angle = Angle(60)
>>> small_angle.degrees
60
>>> small_angle.is_acute()
True
>>> big_angle = Angle(320)
>>> big_angle.is_acute()
False
>>> funny_angle = Angle(1081)
>>> funny_angle.degrees
1
>>> total_angle = small_angle + big_angle
>>> total_angle.degrees
20
```

As you can see, `Angle` keeps track of the angle size, wrapping around so it's in a range of 0 up to 360 degrees. You also have an `is_acute()` method, to tell you if its size is under 90 degrees, and an `__add__()` method for arithmetic.[4]

Suppose this `Angle` class is defined in a file named `angles.py`. Here's how we create a simple test for it—in a separate file, named `test_angles.py`:

```
import unittest
from angles import Angle

class TestAngle(unittest.TestCase):
    def test_degrees(self):
        small_angle = Angle(60)
        self.assertEqual(60, small_angle.degrees)
        self.assertTrue(small_angle.is_acute())
        big_angle = Angle(320)
        self.assertFalse(big_angle.is_acute())
        funny_angle = Angle(1081)
        self.assertEqual(1, funny_angle.degrees)

    def test_arithmetic(self):
        small_angle = Angle(60)
        big_angle = Angle(320)
        total_angle = small_angle + big_angle
        self.assertEqual(20, total_angle.degrees,
                         'Adding angles with wrap-around')
```

As you look over this code, notice a few things:

- There's a class called `TestAngle`. You just define it, but you do not create any instance of it. It subclasses `TestCase`.
- You define two methods, `test_degrees()` and `test_arithmetic()`.
- Both `test_degrees()` and `test_arithmetic()` have assertions, using some methods of `TestCase`: `assertEqual()`, `assertTrue()`, and `assertFalse()`.
- The last assertion includes a custom message as its third argument.

To see how this works, let's define a stub for the `Angle` class in `angles.py`:

```
# angles.py - stub version
class Angle:
    def __init__(self, degrees):
        self.degrees = 0
    def is_acute(self):
        return False
```

4 Remember from Chapter 6; `__add__()` is a magic method which makes binary addition with + work.

```
    def __add__(self, other_angle):
        return Angle(0)
```

This `Angle` class defines all the attributes and methods it is expected to have, but otherwise does nothing useful. We need a stub like this to verify the test can run correctly, and alert us to the fact that the code isn't working yet.

The `unittest` module is used not just to define tests, but also to run them. You do so on the command line like this:[5]

```
python -m unittest test_angles.py
```

Run the test, and you'll see the following output:

```
$ python -m unittest test_angles.py
FF
======================================================
FAIL: test_arithmetic (test_angle.TestAngle)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_angles.py", line 18, in test_arithmetic
    self.assertEqual(20, total_angle.degrees, 'Adding angles with wrap-around')
AssertionError: 20 != 0 : Adding angles with wrap-around

======================================================
FAIL: test_degrees (test_angle.TestAngle)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_angles.py", line 7, in test_degrees
    self.assertEqual(60, small_angle.degrees)
AssertionError: 60 != 0

------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=2)
```

Notice:

- Both test methods are shown. They both have a failed assertion highlighted.
- `test_degrees()` makes several assertions, but only the first one has been run—once it fails, the others are not executed.
- For each failing assertion, you are given the line number, its expected and actual values, and its test method.

5 For running Python on the command line, this book uses `python` for the executable. But depending on how Python was installed on your computer, you may actually need to run `python3` instead. Check by running `python -V`, which reports the version number. If it says 2.7 or lower, that is the legacy version; you want to run `python3` instead.

- The custom message in `test_arithmetic()` shows up in the output.

This demonstrates one useful way to organize your test code: in a single test module (`test_angles.py`), you define one or more subclasses of `unittest.TestCase`. Here, I just define `TestAngle`, containing tests for the `Angle` class. Within this, I create several test methods, for testing different aspects of the class. In each of these test methods, I can have as many assertions as makes sense.

It's traditional to start a test class name with the string `Test`, but that is not required; `unittest` will find all subclasses of `TestCase` automatically. But every method must start with the string `test`. If it starts with anything else (even `Test`), `unittest` will not run its assertions.

Running the test and watching it fail is an important first step. It verifies that the test does, in fact, actually test your code. As you write more and more tests, you'll occasionally create a test; run it, expecting it to fail; and find that it unexpectedly passes. That's a bug in your test code! Fortunately you ran the test first, so you caught it right away.

In the test code, we defined `test_degrees()` before `test_arithmetic()`, but they were actually run in the opposite order. It's important to craft your test methods to be self-contained, and not depend on one being run before the other, for several reasons. One of them is that the order in which they are defined is generally not the order in which they are executed.[6]

(If you find yourself wanting to run tests in a certain order, this might be better handled with `setUp()` and `tearDown()`, explained in the next section.)

At this point, we have a correctly failing test. If I'm using version control and working in a branch, this is a good moment to check in the test code, because it specifies the correct behavior (even if it's presently failing). The next step is to actually make that test pass. Here's one way to do it:

```
# angles.py, version 2
class Angle:
    def __init__(self, degrees):
        self.degrees = degrees % 360
    def is_acute(self):
        return self.degrees < 90
    def __add__(self, other_angle):
        return Angle(self.degrees + other_angle.degrees)
```

6 In current Python versions, the test methods are executed in alphabetical order. This order is fragile, because it changes when you add a new test method.

Now when I run my tests again, they all pass:

```
python -m unittest test_angles.py
..
----------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

`assertEqual()`, `assertTrue()`, and `assertFalse()` will be the most common assertion methods you use, along with `assertNotEqual()`, which does the opposite of `assertEqual()`. `TestCase` provides many others, such as `assertIs()`, `assertIsNone()`, `assertIn()`, and `assertIsInstance()`—along with "not" variants like `assertIsNot()`. Each takes an optional final message-string argument, like "Adding angles with wrap-around" in `test_arithmetic()` above. If the test fails, the message is printed in the output, which can give helpful advice to whoever is troubleshooting a broken test.[7]

If you try checking that two dictionaries are equal, and they are not, the output is tailored to the data type: highlighting which key is missing, or which value is incorrect, for example. This also happens with lists, tuples, and sets, making troubleshooting much easier. What's actually happening is that `unittest` provides certain type-specialized assertions, such as `assertDictEqual()`, `assertListEqual()`, and more. You almost never need to invoke them directly: if you invoke `assertEqual()` with two dictionaries, it automatically dispatches to `assertDictEqual()`, and similar for the other types. So you get this usefully detailed error reporting for free.

The `assertEqual()` lines take two arguments, and I always write the expected, correct value first:

```
small_angle = Angle(60)
self.assertEqual(60, small_angle.degrees)
```

It does not matter whether the expected value is first or second, but it's smart to pick an order and stick with it—at least throughout a single codebase, and maybe for all code you write. Sticking with a consistent order greatly improves the readability of your test output, because you never have to decipher which is which. Believe me, this will save you a *lot* of time; nothing will throw off your momentum more than confusing the expected and actual values for each other, only to realize 20 minutes later you had them mixed up in your head. If you're on a team, negotiate with your teammates to agree on a consistent order, and enforce it.

7 This could be you, months or years down the road. Be considerate of your future self!

Fixtures and Common Test Setup

As an application grows and you write more tests, you will find yourself writing groups of test methods that start or end with the same lines of code. This repeated code—which does some kind of pretest setup, and/or after-test cleanup—can be consolidated in the special methods `setUp()` and `tearDown()`.

Each of your `TestCase` subclasses can define `setUp()`, `tearDown()`, both, or neither. If defined, `setUp()` is executed just before each test method starts; `tearDown()` is run just after. This is repeated for every single test method.

Here's a realistic example of when you might use it. Imagine working on a tool that saves its state between runs in a special JSON file. We'll call this the "state file". On start, your tool reads the state from the file; if the tool has any state change while running, that gets written to the file on exit.

It makes sense to write a class to manage this state file. A stub might look like:

```
# statefile.py - Stub version
class State:
    def __init__(self, state_file_path):
        # Load the stored state data, and save
        # it in self.data.
        self.data = { }
    def close(self):
        # Handle any changes on application exit.
```

In fleshing out this stub, we want our tests to verify the following:

- If I alter the value of an existing key, that updated value is written to the state file.
- If I add a new key-value pair to the state, it is recorded correctly in the state file.
- If I remove a key-value pair from the state, it is also removed in the state file.
- If the state is not changed, the state file's content stays the same.

For each test, we want the state file to be in a known starting state. Afterward, we want to remove that file so that our tests don't leave garbage on the filesystem. Here's how the `setUp()` and `tearDown()` methods accomplish this:

```
import os
import unittest
import shutil
import tempfile
from statefile import State

INITIAL_STATE = '{"foo": 42, "bar": 17}'

class TestState(unittest.TestCase):
    def setUp(self):
```

```
        self.testdir = tempfile.mkdtemp()
        self.state_file_path = os.path.join(
            self.testdir, 'statefile.json')
        with open(self.state_file_path, 'w') as outfile:
            outfile.write(INITIAL_STATE)
        self.state = State(self.state_file_path)

    def tearDown(self):
        shutil.rmtree(self.testdir)

    def test_change_value(self):
        self.state.data["foo"] = 21
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertEqual(21,
            reloaded_statefile.data["foo"])

    def test_add_key_value_pair(self):
        self.state.data["baz"] = 42
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertEqual(42, reloaded_statefile.data["baz"])

    def test_remove_key_value_pair(self):
        del self.state.data["bar"]
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertNotIn("bar", reloaded_statefile.data)

    def test_no_change(self):
        self.state.close()
        with open(self.state_file_path) as handle:
            checked_content = handle.read()
        self.assertEqual(INITIAL_STATE, checked_content)
```

In `setUp()`, you create a fresh temporary directory, and write the contents of `INITIAL_DATA` inside. Since we know each test will be working with a `State` object based on that initial data, we create that object, and save it in `self.state`. Each test can then work with that object, confident that it is in the same consistent starting state, regardless of what any other test method does. In effect, `setUp()` creates a private sandbox for each test method.

The tests in `TestState` would all work reliably with just `setUp()`. But we also want to clean up the temporary files we create; otherwise, they will accumulate over time with repeated test runs. The `tearDown()` method runs after each `test_*` method completes, even if some of its assertions fail. This ensures the temp files and directories are all removed completely.

The generic term for this kind of preparation is called a *test fixture*. A test fixture is whatever needs to be done or set up before a test can properly run. In this case, we set

up the text fixture by creating the state file, and the `State` object. A text fixture can be a mock database, a set of files in a known state, some kind of network connection, or even starting a server process. You can do all these with `setUp()`.

`tearDown()` is for shutting down and cleaning up the text fixture: deleting files, stopping the server process, etc. You will not always need a tear-down, but in some cases it is essential. If `setUp()` starts some kind of server process, for example, and `tearDown()` fails to terminate it, then `setUp()` may not be able to run for the next test.

When you write these method names, the camel-casing matters. People sometimes misspell them as `setup()` or `teardown()`, then wonder why they are not automatically invoked. Any uncaught exception in either `setUp()` or `tearDown()` will cause that test to fail, after which `unittest` immediately skips to the next test. For errors in `setUp()`, this means none of that test's assertions will run (though it still shows as a clear error in the output). For `tearDown()`, the test is marked as failing, even if all the individual assertions passed.

Asserting Exceptions

Sometimes your code is supposed to raise an exception, under certain conditions. If that condition occurs, and your code does *not* raise the correct exception, that's a bug. How do you write test code for this situation?

You can verify that behavior with a special method of `TestCase`, called `assertRaises()`. It's used in a `with` statement in your test; the block under the `with` statement is asserted to raise the exception.

For example, suppose you are writing a library that translates Roman numerals into integers. You might define a function called `roman2int()`:

```
>>> roman2int("XVI")
16
>>> roman2int("II")
2
```

In thinking about the best way to design this function, you decide that passing nonsensical input to `roman2int()` should raise a `ValueError`. Here's how you write a test to assert that behavior:

```
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError):
            roman2int("This is not a valid roman numeral.")
```

If you run this test, and `roman2int()` does *not* raise the error, this is the result:

```
$ python -m unittest test_roman2int.py
F
======================================================
FAIL: test_roman2int_error (test_roman2int.TestRoman)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_roman2int.py", line 7, in test_roman2int_error
    roman2int("This is not a valid roman numeral.")
AssertionError: ValueError not raised

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

When you fix the bug, and `roman2int()` raises `ValueError` like it should, the test passes.

Using Subtests

Sometimes you will want to iterate through many test inputs, to thoroughly validate the input range and cover many edge cases. You could simply write a parade of assert methods, but that becomes tediously repetitive, and more importantly, it will stop with the first failing assertion. Sometimes it is tremendously helpful to run all these assertions so that you have a full picture of which are passing and which are not.

Python's `unittest` library supports this with a feature called *subtests*. This lets you conveniently iterate through a potentially large collection of test inputs, with well-presented (and easy to comprehend) reporting output. Pytest calls its version of this feature *parameterized tests*, which is probably a better name. But since we are focused on `unittest`, we will call them subtests.

Imagine a function called `numwords()`, which counts the number of unique words in a string (ignoring punctuation, spelling, and spaces):

```
>>> numwords("Good, good morning. Beautiful morning!")
3
```

Suppose you want to test how `numwords()` handles excess whitespace. You can easily imagine a dozen different reasonable inputs that will result in the same return value, and want to verify it can handle them all. You might create something like this:

```
class TestWords(unittest.TestCase):
    def test_whitespace(self):
        self.assertEqual(2, numwords("foo bar"))
        self.assertEqual(2, numwords("    foo bar"))
        self.assertEqual(2, numwords("foo\tbar"))
        self.assertEqual(2, numwords("foo   bar"))
        self.assertEqual(2, numwords("foo bar    \t   \t"))
        # And so on, and so on...
```

Seems a bit repetitive, doesn't it? The only thing varying is the argument to `numwords()`. We might benefit from using a `for` loop:

```
    def test_whitespace_forloop(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
            ]
        for text in texts:
            self.assertEqual(2, numwords(text))
```

At first glance, this seems better: more readable and maintainable. If we add new variants, it's just another line in the `texts` list. And if I rename `numwords()`, I only need to change it in one place in the test.

However, using a `for` loop like this creates more problems than it solves. Suppose you run this test and get the following failure:

```
$ python -m unittest test_words_forloop.py
F
======================================================
FAIL: test_whitespace_forloop (test_words_forloop.TestWords)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_forloop.py", line 17, in test_whitespace_forloop
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Look closely, and you'll realize that `numwords()` returned 3 when it was supposed to return 2.

Pop quiz: out of all the inputs in the list, which caused the bad return value?

The way we've written the test, there is no way to know. All you can infer is that *at least* one of the test inputs produced an incorrect value. You don't know which one. That's the first problem.

The second problem—which the original test also suffers from—is that everything stops with the first failed assertion. For this kind of function, knowing all the failing inputs, and the incorrect results they create, would be *very* helpful for quickly understanding what's going on.

Subtests solve these problems. Our `for`-loop solution is actually quite close. All we have to do is add one line. Do you see it below?

```
    def test_whitespace_subtest(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
            ]
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

Just inside the `for` loop, we write `with self.subTest(text=text)`. This creates a context in which assertions can be made, and even fail. Regardless of whether they pass or not, the test *continues* with the next iteration of the `for` loop. At the end, *all* failures are collected and reported in the test result output, like this:

```
$ python -m unittest test_words_subtest.py

======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='foo\tbar')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='foo bar    \t   \t')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 4

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=2)
```

Behold the opulence of information in this output:

- Each individual failing input has its own detailed summary.
- We are told what the full value of `text` was.
- We are told what the actual returned value was, and it is clearly compared to the expected value.
- No values are skipped. We can be confident that these two are the *only* failures.

This is *much* better. The two offending inputs are "`foo\tbar`" and "`foo bar \t \t`". These are the only values containing tab characters, so you can quickly realize the nature of the bug: tab characters are being counted as separate words.

Let's look at these three lines of code again:

```
for text in texts:
    with self.subTest(text=text):
        self.assertEqual(2, numwords(text))
```

The key-value arguments to `self.subTest()` are shown in the reporting output. You can pass in whatever key-value pairs you like; they can be anything that helps you understand exactly what is wrong when a test fails. Often you will want to pass everything that varies from the test cases; here, that is only the string passed to `numwords()`.

Be clear that in these three lines, the symbol `text` is used for two different things. Look at lines 1, 2, and 3 again:

```
for text in texts:
    with self.subTest(text=text):
        self.assertEqual(2, numwords(text))
```

In line 1, the `text` is the same variable that is passed to `numwords()` on line 3. But on line 2, in the call to `subTest()`, you have `text=text`. The left-hand `text` is actually a parameter that is used in the reporting output if the test fails. The right-hand side is the value of that parameter, which, in this case, happens to also be called `text`.

It can clarify if we use `input_text` as the parameter to `subTest()` instead:

```
for text in texts:
    with self.subTest(input_text=text):
        self.assertEqual(2, numwords(text))
```

Then the failure output might look like:

```
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (input_text='foo\tbar')
```

See how at the end of that `FAIL` line, it says `input_text` instead of `text`? That is because we used a different argument parameter when calling `subTest()`. In fact, we can use whatever parameter name we want, but it often works best if we use same identifier name throughout.

Conclusion

Let's recap the big ideas. Test-driven development means we create the test first, along with whatever stubs we need to make the test run. We then run it and watch it fail. *This is an important step. You must run the test and see it fail.*

This is important for two reasons. You don't really know if the test is correct until you verify that it *can* fail. As you write automated tests more and more over time, you will be surprised at how often you write a test and run it, expecting to see it fail, only to discover it passes. As far as I can tell, every good, experienced software engineer still occasionally does this—even after doing TDD for years! This is why we build the habit of always verifying the test fails first.

The second reason is more subtle. As you gain experience with TDD and become comfortable with it, you will find the cycle of writing tests and making them pass lets you get into a state of flow. This means you are enjoyably productive and focused, in a way that is easy to maintain over time. You will get addicted to this.

Is it important that you strictly follow TDD? People have different opinions on this, some of them *very* strong. Personally, I went through a period of almost a year where I followed TDD quite strictly. As a result, I got *very* good at writing tests, and writing them rapidly.

Now that I've developed that level of skill, I find that I follow TDD most of the time, but less often than I did when learning. I have noticed that TDD is most powerful when I have great clarity on the software's design, architecture and APIs; it helps me get into a cognitive state that seems accelerated, so I can more easily maintain my mental focus, and produce quality code faster.

But I find it very hard to write good tests when I don't yet have that clarity—when I am still thinking through how I will structure the program and organize the code. In fact, I find TDD slows me down in that phase, as any test I write will probably have to be completely rewritten several times, if not deleted, before things stabilize. In these situations, I prefer to get a first version of the code working through manual testing, then write the tests afterward.

For this reason, if your particular job requires a lot of exploratory coding—data scientists, I am looking at you—then TDD may not be something you do all the time. If that is the case, there are many benefits to still doing it as much as you can. Remember, this is a superpower. But only if you use it.

No matter your situation, I encourage you to find a way to do strict TDD for a period of time, simply because of what it will teach you. As you develop your skill at writing tests, you can step back and evaluate how you want to integrate it into your daily routine.

CHAPTER 8

Module Organization

For anything more than a small program, you will want to organize your code into *modules*. This is also the unit of organization for reusable libraries—both libraries you create, and outside code you import into your own codebase.

Python's module system is a delight: easy to use, well designed, and extremely flexible. Making full use requires understanding its mechanisms for imports, and how that works with namespacing. We will dive deep into all of that in this chapter.

In particular, we will focus on how modules *evolve*. Requirements change over time; as new requirements come in, and as you get more clarity on existing requirements, and how to best organize your codebase. So we will cover the best practices for refactoring and updating module structure during the development process.

This is an important and practical part of working with modules. But for some reason, it is never talked or written about. Until now.

Spawning a Module

To touch on everything important about modules, we will follow the lifecycle of a small Python script that gradually grows in scope and lines of code—eventually growing to a point where we want to package its components[1] into reusable modules. We do this not just for sensible organization, but also so we can import them into other applications. This evolution from script to spin-off library happens all the time in real software development.

1 A *component* is a general term for some modular unit of software. Simple components can be a function; a class; or an instance of a class. Components can also be larger-scale structures, such as whole libraries or services. But in this chapter, "component" just means "something you can import from a Python module".

Imagine you create a little Python script, called `findpattern.py`. Its job is simple: to scan through a text file, and print out those lines which contain a certain substring.

This is similar to a program called `grep` in the Linux (and Unix) world, which operates on a stream of lines of text, filtering for those which match a pattern you specify. `findpattern.py` is less powerful than `grep`. But it is more portable, and—more importantly—lets us illustrate everything we need in this chapter.

```
# findpattern.py
import sys

def grepfile(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')

pattern, path = sys.argv[1], sys.argv[2]
for line in grepfile(pattern, path):
    print(line)
```

This program defines a generator function called `grepfile()`, which may remind you of the `matching_lines_from_file()` function in Chapter 1—because it demonstrates many of the same Python best practices. It takes two arguments: `pattern`, the substring pattern to search for, and `path`, the path to the file to search in.

Imagine you have a file named `log.txt`, containing simple log messages, one per line, like this:

```
ERROR: Out of milk
WARNING: Running low on orange juice
INFO: Temperature adjusted to 39 degrees
ERROR: Alien spacecraft crashed
```

Your program needs to get these two values from the command line. So you invoke the program like this:

```
% python findpattern.py ERROR log.txt
```

It is best to do extract the command-line arguments with a specialist library for parsing them, such as Python's built-in `argparse` module. But to avoid a side quest explaining how that works, we will simply extract them from `argv` in the `sys` module.

`sys.argv` is a list of strings. Its first element, at index 0, is the name of the program you are running; so its value will be "`findpattern.py`" in this case. What we want are the command-line arguments, *ERROR* and *log.txt*. These will be at indices 1 and 2 in `sys.argv`, respectively. So we load those into the `pattern` and `path` variables. With that out of the way, we can simply iterate through the generator object returned by `grepfile()`, printing out each matching line.

Whenever you create a Python file, that also creates a module of the same name (minus the *.py* extension). So our `findpattern.py` file, in addition to being a complete program, also creates a module named `findpattern`. This happens automatically; you do not have to declare this, or take any extra steps.

Whenever you have a module, that means you can import from it. In particular, we have a nice function named `grepfile()`, which is general enough that we may want to reuse it in other programs. Let's do that.

You decide to create a program called `finderrors.py` to show only the ERROR lines —similar to `findpattern.py`, but more specialized.

`grepfile()` is a good tool for implementing this, so you decide to import it. Your first version looks like this:

```
# finderrors.py
import sys
from findpattern import grepfile

path = sys.argv[1]
for line in grepfile('ERROR:', path):
    print(line)
```

This looks straightforward. But when you run the program, you get a strange error:

```
$ python finderrors.py log.txt
Traceback (most recent call last):
  File "finderrors.py", line 3, in <module>
    from findpattern import grepfile
  File "findpattern.py", line 10, in <module>
    pattern, path = sys.argv[1], sys.argv[2]
IndexError: list index out of range
```

IndexError? What on earth would cause that?

Let's step through the stack trace. You can see that the error originates in line 3 of `finderrors.py`. That is interesting, because it is the import line—where `grepfile()` is imported from the `findpattern` module.

Next, the stack trace descends into the `findpattern.py` file. Specifically, on line 10, where it unpacks variables from `sys.argv`.

Now you understand the problem. `findpattern.py` was written to use as a standalone program. But we are creating a new program and want to reuse code. In the process of importing, `sys.argv` is unpacked, expecting two arguments, instead of the one argument which `finderrors.py` actually takes.

When you import from a module, Python creates that module by executing its code. It executes the `def` statements to create the functions, and executes the `class`

statements to create the classes. But in the process of creating the module, Python executes the entire file—including that `sys.argv` line.

So what do we do? The code is correct for running `findpattern.py`. But it doesn't allow `finderrors.py` to run. How do we import from `findpattern` and allow both programs to run?

The solution is to use a *main guard*. It looks like this:

```
# Replace the final 3 lines of findpattern.py with this:
if __name__ == "__main__":
    pattern, path = sys.argv[1], sys.argv[2]
    for line in grepfile(pattern, path):
        print(line)
```

We have taken the last three lines and indented them inside an "`if`" block. The "`if`" condition references a magic variable called `__name__`. This variable is automatically and globally defined, and will always have a string value. But to understand the nature of that value, we must first understand something about how Python programs are executed.

When you execute a Python program, often the code which makes up the program is spread across more than one file. That is what happens here, right? When you run `finderrors.py`, some of this program's code is in that file. But some of the code is in the `findpattern.py` file. So the Python code for this program is distributed over two different files.

But here's the important point: whenever you run a Python program consisting of several files, one of those will be the *main executable* for the program. That is the file which will show in the process table or task manager of the operating system. When you run the `finderrors.py` program, then `finderrors.py` is the main executable. That is what will show up in the process table, even though it is also utilizing code in the `findpattern.py` file. Of course, real Python programs often use dozens, hundreds, or even thousands of distinct Python files, especially when your program uses many libraries.

And so, back to `__name__`. This magic variable will have one of two values:

- If it is referenced inside the main executable file, its value will be the string "`__main__`".
- If it is referenced in any other Python file, it will be the name of the *module* that file creates.

And so we use this in `findpattern.py`, to make this file usable as a stand-alone program, *and* as a module which can be imported from. By checking the value of `__name__`, we effectively partition the file into two parts: code which is always

executed (and thus its objects are importable), and code which is executed only when this file is itself run as a program.

Here is the full source of our amended `findpattern.py`:

```
import sys

def grepfile(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')

if __name__ == "__main__":
    pattern, path = sys.argv[1], sys.argv[2]
    for line in grepfile(pattern, path):
        print(line)
```

Let's step through this. Suppose you execute `findpattern.py` as the program. Then the `findpattern.py` file is the main executable, so the value of `__name__` will be equal to "`__main__`". This means the "`if`" condition evaluates to `True`, and the final three lines are executed.

In contrast, imagine you run `finderrors.py`. This file imports from the `findpattern` module, which means all lines in the `findpattern.py` file are executed. But the value of `__name__` is set to the string "`findpattern`". So the "`if`" condition evaluates to `False`, and the final three lines are skipped. Now both programs work:

```
$ python findpattern.py ERROR log.txt
ERROR: Out of milk
ERROR: Alien spacecraft crashed
$ python finderrors.py log.txt
ERROR: Out of milk
ERROR: Alien spacecraft crashed
```

Creating Separate Libraries

Time passes, and new requirements roll in—as they always do. You now need a variant of the `grepfile()` function, which is case insensitive. Let's call it `igrepfile()`.

In your team, suppose you are in charge of `finderrors.py`, and your coworker is in charge of `findpattern.py`. You are importing the `grepfile()` function from `findpattern.py`. But where are you going to put the new `igrepfile()` function?

You cannot put it in `findpattern.py`. Your coworker is politely allowing you to reuse his code by importing from the `findpattern` module, but he draws the line at adding new functions he does not care about to his file. That is just creating more maintenance work for him.

At this point, it makes more sense to organize the code into a different module. Up to now, `findpattern.py` has been filling two completely different roles. It's a program that does something useful, *and* it is a container of sorts: holding components that can be reused by other programs—the `grepfile()` function, in particular.

Now you are going to separate these roles. You create a new file, called `greputils.py`. It is not meant to be an executable program. Its sole purpose is to create a module, called `greputils`, holding reusable code.

In particular, it will define the `grepfile()` function. You cut the entire definition of `grepfile()` from `findpattern.py`, and paste it into `greputils.py`. Then in both `findpattern.py` and `finderrors.py`, you add this import statement:

```
from greputils import grepfile
```

Congratulations: you have created a reusable library for your team's software.

This is a great step forward in code organization. You can stuff whatever new classes and functions make sense in this `greputils` module…without molesting any other program which relies on it. A clear separation of roles.

Notice something else: this has evolved naturally. The entire process described above is fully realistic, in terms of how a module of reusable code "emerges" during normal application development. This example shows how it happens in a team, but it happens much the same way when you are developing solo. As you develop new programs, you naturally find you want to reuse functions and classes from your previous projects, and it only makes sense to collect them all in a single convenient module.

The current versions of our files:

```
# greputils.py
def grepfile(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')
```

```
# findpattern.py
import sys

from greputils import grepfile
pattern, path = sys.argv[1], sys.argv[2]
for line in grepfile(pattern, path):
    print(line)
```

```
# finderrors.py
import sys
from greputils import grepfile

path = sys.argv[1]
```

```
for line in grepfile('ERROR:', path):
    print(line)
```

Continuing to the next phase of our module's evolution, imagine you keep adding new functions and classes over time. So the `greputils.py` file gets bigger and bigger, with more and more lines of code.

To make it concrete: imagine you often have to check whether a text file contains a certain substring. You just want a yes or no answer to that question; you don't need the set of lines that match, you just want to know whether at least one line matches, or none of them do.

This function should return `True` if the text file contains that string, and `False` otherwise. You call this function `contains()`:

```
def contains(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                return True
    return False
```

You also create a case-insensitive version named `icontains()`, and so on; you continue creating new functions and classes over time as you and your teammates need them, always adding them to `greputils.py`.

Eventually, this file will just get too big. It is going to be awkward to work with at best, and encourage excessive intercoupling and code entanglement at worst. If there are multiple developers hacking the same codebase, it may even increase the frequency of merge conflicts. And code logically separated into different files is just easier to work with, as separate tabs in IDEs.

So we refactor the module. Make this distinction: the module is *not* the file `greputils.py`. Rather, the module is `greputils`, which happens to be *implemented* as a file named `greputils.py`. But there are other ways to implement the same module, with the same contents and the exact same interface for importing.

Specifically, we can implement the module as a collection of several files, organized in a particular way. How do you do that?

Multifile Modules

The first step is to create a directory, with the name of the module: `greputils`. And this directory will contain several files.

The first is a file named `__init__.py`. Just like the method for Python classes, except with *.py* at the end. This is where we create the interface to the module, in terms of what you can import from it directly.

But in order to do that, we first create another file in the `greputils` folder, which we will call `files.py`. This is where we will put the file-grepping functions: `grepfile()`, `igrepfile()`, and any others we have created so far. We will also create a third file, named `contain.py`, which is where we put the "does this file contain this string" functions, like `contains()` and `icontains()`. Now, our filesystem layout looks like this:

```
greputils/
greputils/__init__.py
greputils/files.py
greputils/contain.py
```

This solves our giant file problem. We can put as many files in this `greputils` folder as we want. No matter how many classes and functions we invent, we can split them up into reasonably sized files, each defining a submodule within the main `greputils` module.

But we have one more step, which returns us to the `__init__.py` file. Remember I said this file creates the import interface of the module. Originally, when everything lived in a single `greputils.py` file, you could write code like:

```
# used by findpattern.py
from greputils import grepfile

# used by a different program
from greputils import icontains
```

When you refactor a function, you want its interface and outward behavior to be unchanged. Your modified implementation should be invisible to anyone using that function. A close analogy holds for modules. When we "refactor" it to be a directory rather than a single file, we want these import statements to continue to work just as they did before.

When you implement a module as a single file, people can automatically import anything in the module with a simple `from MODULENAME import ...` statement. But when you implement a module as a directory, that is not automatic. You must explicitly declare what components will be directly importable in this way.

You do that in the `__init__.py` file. To see how it works, we must understand a new concept, called *submodules*.

When you implement a module as a directory, the files in that directory create *submodules*. These are accessed in a hierarchy under the top-level module, using a dotted-name syntax. For example, if `grepfile()` is in the `greputils/files.py` file, then you can import it with a statement like:

```
from greputils.files import grepfile
```

But we do not want to break all our existing import statements. Even if we did not mind the extra typing. The solution relies on the fact that anything inside `greputils/__init__.py` will be directly importable from the module itself. So all we have to do is import the submodule contents into the `__init__.py` file. One way is to do it like this:

```
# inside greputils/__init__.py
# (Spoiler: Don't do it like this, there is a better way).
from greputils.files import grepfile
```

That will actually work. But it is not as modular as it could be. What if you change the name of the module in the future? This creates a new place we need to change it, or miss it and create a bug. Or maybe we want to reorganize the module in some other way, which creates the same problem.

Instead, do a *relative import*. Inside the `__init__.py` file, write this:

```
# inside greputils/__init__.py
from .files import grepfile
```

So you are importing from `.files`, not `greputils.files`. That leading "`.`" is important here. This is what makes it a relative import, and its syntax only works inside a module that is implemented as a directory. When you use it inside the `__init__.py` file, it will import objects from sub-modules relative to that directory, and make them accessible inside the `__init__.py` file.

We do all this because anything in the `__init__.py` file is directly importable from the module itself. In other words, because of these relative imports, this statement works again:

```
from greputils import grepfile
```

We do this for all components for all the submodules. Our final `__init__.py` file, including everything we have created so far, looks like this:

```
# greputils/__init__.py
from .files import grepfile, igrepfile
from .contain import contains, icontains
```

Now we can import all these functions—`grepfile()`, `igrepfile()`, `contains()`, and `icontains()`—directly:

```
# In one program...
from greputils import contains, grepfile

# And in another:
from greputils import icontains, igrepfile
```

You can be selective in what you import into the `__init__.py` file. Your submodules may contain internal helper functions and classes which you do not necessarily want

other people importing, using, and crafting their code to depend on. This is likely to be common, in fact, as your modules grow beyond a certain size.

Handling this is easy: simply do not import them into the `__init__.py` file. Then they are not directly importable from the module. Someone can still import them from the dotted path of the submodule if they peek into the source to find they are there, but few people will do that. And if you change one of those internal components later in a way that breaks their code, that is arguably their problem and not yours.

Import Syntax and Version Control

When importing multiple components, you can do it all on one line, with a single import statement. The `greputils/__init__.py` does this, importing from its submodules:

```
# greputils/__init__.py
from .files import grepfile, igrepfile
from .contain import contains, icontains
```

If you are importing a couple of items, that works fine. But suppose you are importing more. Putting all those on one line creates problems, especially if you are working in a team.

Imagine you and I are developing the same codebase, in different feature branches. In your branch, you add a new function named `grepfileregex()`. You put this in `greputils/files.py`, and modify `greputils/__init__.py` like this:

```
from .files import grepfile, igrepfile, grepfileregex
```

In my branch, I rename `igrepfile()` to `grepfilei()`. And edit the `__init__.py` file like this:

```
from .files import grepfile, grepfilei
```

When we finish our work in the branch, and merge into main, suddenly we have a race condition. Whoever merges first will have no problem. But the one who merges last will get a merge conflict. Our current version-control tools do not know how to resolve this manually; the loser of this race condition has to manually clean it up. It is time-consuming at best, and it risks creating new bugs at worst.

There is an easy solution. Python allows you to split imported components over several lines, using this syntax:

```
from .files import (
    grepfile,
    igrepfile,
)
from .contain import (
```

```
    contains,
    icontains,
)
```

Note the parentheses. This is much better, because our version control software can resolve the merge automatically, without the risk of introducing new bugs. As a general rule, if you are importing more than one item, your life will be happier if you split the import over several lines in this way.

Notice another detail here. Just looking at the first relative import:

```
from .files import (
    grepfile,
    igrepfile,
)
```

Do you see how there is a comma after "`igrepfile`", even though it is the last item in the sequence? Python does not *require* you to put a comma there. But I recommend you do, because it pinpoints the diffs even further.

Imagine you do not put that comma there, then add your new `grepfileregex()` function. So you are changing it from this:

```
# no ending comma
from .files import (
    grepfile,
    igrepfile
)
```

…to this:

```
from .files import (
    grepfile,
    igrepfile,
    grepfileregex
)
```

You are adding a line for `grepfileregex()` but also modifying the previous line—adding a comma after `igrepfile`. So the diff will delete the old line without the comma, and add two lines. Like this:

```
-    igrepfile
+    igrepfile,
+    grepfileregex
```

In contrast, if you append a comma after every imported item, you are changing it from this:

```
# WITH an ending comma
from .files import (
    grepfile,
    igrepfile,
)
```

…to this:

```
from .files import (
    grepfile,
    igrepfile,
    grepfileregex,
)
```

This means your diff goes from three lines down to just one:

```
+    grepfileregex,
```

Like I said: more pinpointed diffs. Only good can come from that. Not only will you have fewer merge conflicts. If your team is doing code reviews, the reviewer will have to think less to decipher what you are changing. They are less likely to miss a bug that would otherwise be easy for them to catch.

Nested Submodule Structure

This directory structure for modules is recursive. A module can be implemented as a file, or as a directory; and this also applies to submodules, sub-submodules, and so on.

Imagine you add a new submodule, called `greputils.net`, collecting functions that check for contents of URLs over a network. At first, `greputils.net` does not have many functions or classes. So this submodule is implemented as a single file, `net.py`, in the `greputils` folder.

But over time, you add something like this:

```
greputils/
greputils/__init__.py
greputils/files.py
greputils/contain.py
greputils/net/__init__.py
greputils/net/html.py
greputils/net/text.py
greputils/net/json.py
```

At this point, you have a choice to make about the module interface—which will be somewhat dictated by how other code is using the `greputils.net` components already. In the first choice, you will import components into `greputils/net/__init__.py`, like this:

```
# in greputils/net/__init__.py
from .html import (
   grep_html,
   grep_html_as_text,
)
from .json import (
   grep_json,
```

```
    grep_json_many,
)
```

This makes each of these functions importable from the `greputils.net` submodule. So in the top-level `__init__.py` file, we can simply do a relative import:

```
# in greputils/__init__.py
from .net import (
    grep_html,
    grep_html_as_text,
    grep_json,
    grep_json_many,
)
```

With this organization, you can import the `grep_html()` function from `greputils` directly, but also from the `greputils.net` submodule:

```
# This...
from greputils import grep_html
# Or this.
from greputils.net import grep_html
```

Both of these will successfully import the function. Whether you *want* people to do both is a different question.

In this case, we started with a file named `greputils/net.py`, which means everything was originally importable from `greputils.net`. For this reason, there may be existing code which does that, in any program which uses `greputils`. By arranging these two `__init__.py` files in this way, we allow that code to continue working unmodified, while also allowing `grep_html` to be imported from `greputils` directly.

We have another choice. We could instead import from the most nested submodules, directly into the top-level `__init__.py` file, like this:

```
# in greputils/__init__.py
from .net.html import (
    grep_html,
    grep_html_as_text,
)
from .net.json import (
    grep_json,
    grep_json_many,
)

# Plus the other relative imports from other submodules.
```

See how this is different? Rather than importing `grep_html()` from `.net`, it is instead imported from `.net.html`—and similar for the others.

So if you do it this way, what goes in `net/__init__.py`? Nothing. In this case, you can make that file empty. The consequence is that `from greputils.net import`

`grep_html` will not work; you can only import it from `greputils`. But if that would not break any existing code, then you may decide there is no reason to support anything but the top-level import. This is certainly an option if you are creating the full module with a nested file structure from the start.

You might wonder: if `net/__init__.py` is empty, is it necessary? No, it is not. Modern Python allows you to omit an empty `__init__.py` file entirely. So the file list will look like this:

```
greputils/
greputils/__init__.py
greputils/files.py
greputils/contain.py
greputils/net/html.py
greputils/net/text.py
greputils/net/json.py
```

See how there is only one `__init__.py` file, at the top-level module directory.

Notice that throughout this process, from the very start, the interface to `greputils` did not change. Back when we had a single `greputils.py` file, you could write `from greputils import grepfile`. And you can write the same thing now, when `greputils` is a directory instead of a file.

In other words, whether you implement your module as a single file or as a directory is just an implementation detail. The developer using your code doesn't have to know or care how you made the module. When you make the change, people using the module probably won't even notice.

This evolution is quite common. It happens just as I have described in realistic software development cycles.

Antipattern Warning

There is an antipattern I have to tell you about. It looks like this:

```
from some_module import *
```

You are using a "`from...import`" statement, but the final field is not a sequence of components to import. Instead, it is the literal "`*`" character. For `greputils`, it would look like this:

```
from greputils import *
```

What this does is import *every single object* from the module. Every function, every class, every global variable. It drops them all into your current namespace. Like dumping a bucket of smelly fish all over the floor.

This is a problem for several reasons. But the worst reason is that it creates a time bomb. Suppose your application uses `greputils`, and also another module called `filesearch`. And imagine you write these two import lines:

```
from greputils import *
from filesearch import *
```

You know that `greputils` has a function called `grepfile()`. And let's say the `filesearch` module does not contain anything with this name, so there is no conflict. You test your code, it works great, and you deploy it. Everything is great.

Now imagine several months pass, and `filesearch` has a critical security update. So without you even knowing, someone in your organization—on the DevOps team, or another developer—decides to upgrade `filesearch`. But this update also introduces a new function, called `grepfile()`—which is *completely unrelated* to the `grepfile()` function in `greputils`, even though it has the same name.

What does Python do? Because the import from `filesearch` comes second, Python will *silently override* the `greputils` version. In other words, when you call `grepfile()` in your program, you are suddenly calling `filesearch.grepfile()`, not `greputils.grepfile()`.

What happens next? If you are lucky, this will cause an immediate and obvious error. Unit tests will fail, or a manual test will catch it before you deploy.

If you are not lucky, you have a time bomb.

In this case, the code path using `grepfile()` is not immediately triggered. It gets deployed to production, and everything appears to work correctly. And the application may continue to run fine just long enough for everyone to forget about the library upgrade.

Until, when you least expect it, that code path containing `grepfile()` finally runs. Boom.

You may not be this unlucky. But if you are, this is just about the worst kind of bug. It is unpredictable and disruptive. And it can be hard to fix simply because it is far enough removed in time that the probable cause is no longer fresh in anyone's mind.

You can get other problems from importing star, but in my opinion this is the worst one. The only protection is to not do it at all.

So what do you do instead? After all, when someone imports star, they're not *trying* to ruin your life. Normally they do it because the module has many functions and classes they need to use.

There are several strategies. If you only need to import a few components, just import each by name. Just like we have been doing:

```
from greputils import grepfile

# And then later in your code:
grepfile("pattern to match", "/path/to/file.txt")
```

This has the advantage of being completely precise and specific. But it is inconvenient when you are importing more than a few items.

An alternative: simply import the module itself. This is just like importing star, except everything is namespaced inside the module. So you get none of the name conflict problems. Like this:

```
import greputils

# And then later in your code:
greputils.grepfile("pattern to match", "/path/to/file.txt")
```

The downside is that you have to type the module name over and over. To alleviate this, Python lets you abbreviate the module name when you import it, with an "`as`" clause. For example, we can rename `greputils` to `gu` inside your code:

```
import greputils as gu

# And then later in your code:
gu.grepfile("pattern to match", "/path/to/file.txt")
```

It is like creating a more convenient alias for the module, for use just inside your program. This is commonly used in different Python library ecosystems:

```
# Data scientists are smart enough to use this a lot.
import numpy as np
import pandas as pd
```

Then in your code, you can refer to `np.array`, `pd.DataFrame`, and so on.

This renaming trick works not just with modules, but also with items imported from a module. So if you are calling a function with a long name over and over, you can give it a shorter or better name:

```
# Import a function and rename it:
from greputils import igrepfile as cigrep

# Similar to:
from greputils import igrepfile
cigrep = igrepfile
```

This is not just useful for making function names shorter. It also can improve clarity. For example, the tremendously useful `dateutil` library provides a function that can automagically parse just about any date-time string you give it, returning a nice `datetime.datetime` instance:

```
>>> from dateutil.parser import parse
>>> parse('Sat Aug 10 08:03:50 2074')
datetime.datetime(2074, 8, 10, 8, 3, 50)
>>> parse('01-05-2081 11:39pm')
datetime.datetime(2081, 1, 5, 23, 39)
>>> parse('5/15/57 22:29')
datetime.datetime(2057, 5, 15, 22, 29)
```

That is extremely useful. But `parse()` is almost the most generic name you can give to a function. It just does not give enough of a clue what the function actually does. You might as well call it `compute()`.

So what I do is rename the function when I import it:

```
from dateutil.parser import parse as parse_datetime
```

Then my code can invoke it with the much more informative name of `parse_datetime()`:

```
>>> parse_datetime('Sat Aug 10 08:03:50 2074')
datetime.datetime(2074, 8, 10, 8, 3, 50)
>>> parse_datetime('01-05-2081 11:39pm')
datetime.datetime(2081, 1, 5, 23, 39)
>>> parse_datetime('5/15/57 22:29')
datetime.datetime(2057, 5, 15, 22, 29)
```

While renaming on import can be useful, it is also possible to overdo it, to the point the code becomes harder to read. As a general rule, I recommend you use this feature to improve readability, and otherwise simply use the original name.

Import Side Effects

Normally, the files of your module will contain definitions of classes and functions, and perhaps assign top-level variables. They might also import from sub-modules which follow the same pattern.

Modules constructed this way have no side effects of code execution during the import itself. But that does not mean there is no execution of code. In fact, that is exactly what happens every time you import.

Few people understand this important point. It relates to what we discussed near the start of this chapter, when we learned about the concept of a "main guard". Imagine you have a module with this code (using `...` as a placeholder for code I am omitting here):

```
CSV_FIELDS = [
    'date',
    'revenue',
    ...
]
```

```
class MissingContent(Exception):
    ...

def extract_params(text):
    ...

class UserData:
    def __init__(self, first_name, last_name, email):
        ...
```

This module provides several components:

- A list of strings, called `CSV_FIELDS`
- An exception called `MissingContent`
- A function called `extract_params()`
- A class called `UserData`

All of these are objects. Yes, even the classes and the function, because in Python everything is an object. How were these objects created? They were created because Python executes the lines of code which define them.

To put it another way, consider these lines:

```
class UserData:
    def __init__(self, first_name, last_name, email):
        ...
```

This is a `class` statement. Python will *execute* this class statement. Executing this statement will create a new object, called `UserData`, which is a class. This will be inserted into the current scope, so lines of code after this `class` statement can create instances of the class. Lines of code prior to this statement cannot, because the `class` statement which created the class object has not been executed yet.

Everything said applies to function definitions, too:

```
def extract_params(text):
    ...
```

This `def` statement is Python code which is executed. The effect of that execution is to create a function object, named `extract_params()`. After that statement is executed, other Python code can invoke that function.

It is import to understand there is no special difference between class and function definitions, versus "normal" code. In order to create the class or function object, Python must execute the `class` or `def` statements. At this level, it is exactly the same as a statement like `x = 1`, as far as Python is concerned.

So when Python imports a module, it must execute that module's code first. But this is "all or nothing". When you import a class, Python does not scan through the code to

find just that `class` statement, and only execute that. It executes the whole module.[2] That is why we need to use main guards (i.e., the `if __name__ == "__main__"` trick).

Some modules exploit this to do import-time code execution on purpose. This is typically used for some kind of initialization or side effect that will take place when that specific module is first imported. For example, imagine a module which creates a database connection object at the top-level:

```
# database.py

# Function to create a new network connection to the DB
def initiate_database_connection():
    ...

# Go ahead and create a connection handle
# This will be a top-level object,
# importable from the module.
conn = initiate_database_connection()
```

This is sometimes useful or necessary. But I suggest you avoid import-time side effects, unless you have a compelling reason.

For one, these import-time side effects are often surprising for people who reuse your code. That can include you, months down the road after you've written the module, and have completely forgotten that importing it will trigger some kind of microservice connection, for example. Another way of saying this: import-time side effects violate the Principle of Least Surprise.

Another problem: you cannot control the exact timing. Python will execute that module code at some point. But *when* is not defined by the language. It is hard to predict what order Python will execute individual module files. And the order can change without warning as you evolve the code, or when you upgrade to a new version of Python. Imagine a program where lines could change order each time you run it; import-time side effects are a bit like that.

Finally, they are inflexible. You cannot avoid executing that code if you do not want to; your only option is to not use the module at all. You generally cannot customize its behavior, because the import itself triggers the execution with hard-coded arguments.

The database example above is guilty of all these crimes. It's a good example of what not to do.

2 Could Python be designed to simply execute the relevant class statement only? Not really. Python is so dynamic as a language, that a class which is defined early in the file can be renamed by a line near the end of that file. The only way to ensure the correct class is imported is to execute the whole module.

That said, creating import-time side effects is sometimes necessary—or at least useful enough that it is worth the downsides. If you encounter that situation, do not worry about it; go ahead and do it. But always ask if you can organize your application to avoid it.

Conclusion

Python's module system is so nicely designed that you can go far with it just by understanding a few basics. And having read this chapter, you know there is a lot more depth. That simple interface over complex and powerful semantics makes it possible to amplify the impact of the code you write. You create functions and classes which solve hard problems that people care about, package them in a nice module interface, and suddenly you have empowered a lot of folks to do more with less effort. It is a form of leverage. And whether you are distributing an open source project, creating a module for your team, or even just packaging your code to make it easier for you to reuse in the future, this investment of your effort always comes back to you in positive ways.

CHAPTER 9

Logging in Python

Logging is critical in many kinds of software. For long-running software systems, it enables continuous telemetry and reporting. And for *all* software, it can provide priceless information for troubleshooting and postmortems. The bigger the application, the more important logging becomes. But even small scripts can benefit.

Python provides logging through the `logging` module. In my opinion, this module is one of the more technically impressive parts of Python's standard library. It's well-designed, flexible, thread safe, and richly powerful. It's also complex, with many moving parts, making it hard to learn well. This chapter gets you over most of that learning curve, so you can fully benefit from what `logging` has to offer. The payoff is well worth it and will serve you for years.

Broadly, there are two ways to use `logging`. One, which I'm calling the *basic interface*, is appropriate for scripts—meaning, Python programs that are small enough to fit in a single file. For more substantial applications, it's typically better to use *logger objects*, which give more flexible, centralized control, and access to logging hierarchies. We'll start with the former, to introduce the key ideas.

The Basic Interface

Here's the easiest way to use Python's logging module:

```
import logging
logging.warning('Look out!')
```

Save this in a script and run it, and you'll see this printed out:

```
WARNING:root:Look out!
```

You can do useful logging right away, by calling functions in the `logging` module itself. You invoked `logging.warning()`, and the output line started with `WARNING`. You can also call `logging.error()`, which gives a different prefix:

```
ERROR:root:Look out!
```

Log Levels

We say that `warning` and `error` are at different *message log levels*. You have a spectrum of log levels to choose from, in order of increasing severity:[1]

debug
: Detailed information, typically of interest only when diagnosing problems or during development.

info
: Confirmation that things are working as expected.

warning
: An indication that something unexpected happened, or indicative of some problem in the near future (e.g., "disk space low"). The software is still working as expected.

error
: Due to a more serious problem, the software has not been able to perform some function.

critical
: A serious error, indicating that the program itself may be unable to continue running.

You use them all just like `logging.warning()` and `logging.error()`:

```
logging.debug("Small detail. Useful for troubleshooting.")
logging.info("This is informative.")
logging.warning("This is a warning message.")
logging.error("Uh oh. Something went wrong.")
logging.critical("We have a big problem!")
```

Each has a corresponding uppercased constant in the library (such as `logging.WARNING` for `logging.warning()`). Use these when defining the *log level threshold*. Run the above, and here is the output:

1 These beautifully crisp descriptions, which I cannot improve upon, are largely taken from the Logging HOWTO documentation.

```
WARNING:root:This is a warning message.
ERROR:root:Uh oh. Something went wrong.
CRITICAL:root:We have a big problem!
```

Where did the debug and info messages go? As it turns out, the default logging threshold is `logging.WARNING`, which means only messages of that severity or greater are actually generated; the others are ignored completely. The list of log levels above is in order of increasing severity; `debug` is considered strictly less severe than `info`, and so on. Change the log level threshold using the `basicConfig()` function:

```
logging.basicConfig(level=logging.INFO)
logging.info("This is informative.")
logging.error("Uh oh. Something went wrong.")
```

Run this new program, and the `INFO` message gets printed:

```
INFO:root:This is informative.
ERROR:root:Uh oh. Something went wrong.
```

Again, the order is `debug()`, `info()`, `warning()`, `error()`, and `critical()`, from lowest to highest severity. When we set the log level threshold, we declare that we only want to see messages of that level or higher. Messages of a lower level are not printed. When you set `level` to `logging.DEBUG`, you see everything; set it to `logging.CRITICAL`, and you only see critical messages; and so on.

The phrase "log level" means two different things, depending on context. It can mean the severity of a message, which you set by choosing which of the functions to use—`logging.warning()`, etc. Or it can mean the threshold for ignoring messages, which is signaled by the constants: `logging.WARNING`, etc.

You can also use the constants in the more general `logging.log` function—for example, a debug message:

```
logging.log(logging.DEBUG,
    "Small detail. Useful for troubleshooting.")
logging.log(logging.INFO, "This is informative.")
logging.log(logging.WARNING, "This is a warning message.")
logging.log(logging.ERROR, "Uh oh. Something went wrong.")
logging.log(logging.CRITICAL, "We have a big problem!")
```

This lets you control the log level dynamically, at runtime. For example:

```
def log_results(message, level=logging.INFO):
    logging.log(level, "Results: " + message)
```

Why Do We Have Log Levels?

If you haven't worked with similar logging systems before, you may wonder why we have different log levels, and why you'd want to control the filtering threshold. It's easiest to see this if you've written Python scripts with repeated calls to `print()`—

including some that are useful for diagnosis when something goes wrong, but a distraction when everything is working fine.

The fact is, some of those `print()`'s are more important than others. Some indicate mission-critical problems that you always want to know about—possibly to the point of waking up an engineer so that they can deploy a fix immediately. Some are important, but can wait until the next work day. Some are details which may have been important in the past, and might be in the future, so you don't want to remove them; in the meantime, they are just line noise.

Log levels help you solve these problems. As you develop and evolve your code over time, you continually add new logging statements of the appropriate severity. You now even have the freedom to be proactive. With "logging" via `print()`, each log statement has a cost—certainly in signal-to-noise ratio, and also potentially in performance. So you might debate whether to include that print statement at all.

But with `logging` you can insert `info` messages, for example, to log certain events occurring as they should. In development, those `INFO` messages help you verify that certain things are happening. In production, you may not want to have them cluttering up the logs, so you just set the threshold higher. If you are doing some kind of monitoring in production, and temporarily need that information, you can adjust the log level threshold to output those messages; when you are finished, you can adjust it back to exclude them again.

When troubleshooting, you can liberally introduce `debug`-level statements to provide extra detailed statements. When done, you simply adjust the log level to turn them off. You can leave them in the code with barely any effect on performance, eliminating any risk of introducing more bugs when you go through and remove them. This also leaves them available if they are needed in the future.

Configuring the Basic Interface

You can change the log-level threshold by calling a function called `basicConfig()`:

```
logging.basicConfig(level=logging.INFO)
logging.debug("You won't see this message!")
logging.error("But you will see this one.")
```

If you use it at all, `basicConfig()` must be called exactly once,[2] and it must happen before the first logging event. (Meaning, before the first call to `debug()`, or

2 You can call it more than once, but anything beyond the first call has no effect. No error is raised, so you need to be careful that you call it exactly one time.

`warning()`, etc.) Additionally, if your program has several threads, `basicConfig()` must be called from the main thread—and *only* the main thread.[3]

You've already met one of the configuration options: `level`. Some of the other options include:

`filename`
: Write log messages to the given file, rather than stderr.

`filemode`
: Set to `"a"` to append to the log file (the default), or `"w"` to overwrite.

`format`
: The format of log records.

By default, log messages are written to standard error. You can also write them to a file—one per line—to be easily read later. Do this by setting `filename` to the log file path. By default it appends log messages, meaning that it will only add to the end of the file if it isn't empty. If you'd rather the file be emptied before the first log message, set `filemode` to `"w"`. Be careful about doing that, of course, because you can easily lose old log messages if the application restarts:

```
# Wipes out previous log entries when program restarts
logging.basicConfig(filename="log.txt", filemode="w")
logging.error("oops")
```

The other value you can pick is `"a"`, for append—that's the default, and it will probably serve you better in production. `"w"` is generally better during development, though. I have wasted many hours of my life wondering why I was seeing a message indicating a bug in the log file, only to realize it came from two edits ago, before I had fixed the bug. Now, I set up my development environment to wipe the logs with every run.

`format` defines what chunks of information the final log record will include, and how they are laid out. These chunks are called *attributes* in the `logging` module docs. One of these attributes is the actual log message—the string you pass when you call `logging.warning()`, and so on. Often you will want to include other attributes as well. Consider the kind of log record we saw above:

```
WARNING:root:Collision imminent
```

This record has three attributes, separated by colons. First is the log level name, and the last is the actual string message you pass when you call `logging.warning()`. In

3 These restrictions are not in place for logger objects, described later.

the middle is the name of the underlying logger object. `basicConfig()` uses a logger called "`root`"; we'll learn more about that later.

You specify the layout you want by setting `format` to a string that defines certain named fields, according to percent-style formatting. Three of them are `levelname`, the log level; `message`, the message string passed to the logging function; and `name`, the name of the underlying logger. Here's an example:

```
logging.basicConfig(
    format="Log level: %(levelname)s, msg: %(message)s")
logging.warning("Collision imminent")
```

If you run this as a program, you get the following output:

```
Log level: WARNING, msg: Collision imminent
```

The default formatting string is:

```
%(levelname)s:%(name)s:%(message)s
```

You indicate named fields in percent-formatting by `%(FIELDNAME)X`, where "`X`" is a type code: `s` for string, `d` for integer (decimal), and `f` for floating-point.

Many other attributes are provided, if you want to include them. Table 9-1 shows a select few from the full list.[4]

Table 9-1. LogRecord attributes

Attribute	Format	Description
asctime	`%(asctime)s`	Human-readable date/time
funcName	`%(funcName)s`	Name of function containing the logging call
lineno	`%(lineno)d`	The line number of the logging call
message	`%(message)s`	The log message
pathname	`%(pathname)s`	Full pathname of the source file of the logging call
levelname	`%(levelname)s`	Text logging level for the message (*DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*)
name	`%(name)s`	The logger's name

You might be wondering why log record format strings use Python 2's ancient percent-formatting style, even in modern Python. As it turns out, backward-compatibility reasons made percent-formatting the only practical choice for the logging module, even after the Python 3 reboot.

4 *https://docs.python.org/3/library/logging.html#logrecord-attributes*

If you want to use the newer string formatting anyway, it is certainly possible.[5] But doing so is complicated enough, and has enough landmines, that it may not be worth the effort. I recommend you simply use percent-formatting with your Python logging.

Passing Arguments

You'll often want to include some kind of runtime data in the logged message. Specify the final log message like this:

```
num_fruits = 14
fruit_name = "oranges"
logging.info(
    "We ate %d of your %s. Thanks!",
    num_fruits, fruit_name)
```

The output:

```
INFO:root:We ate 14 of your oranges. Thanks!
```

We call `info()` with three parameters. First is the format string; the second and third are arguments. The general form is:

```
logging.info(format, *args)
```

You can pass zero or more arguments, so long as each has a field in the format string:

```
# Do it like this:
logging.info("%s, %s, %s, %s, %s, %s and %s",
    "Doc", "Happy", "Sneezy", "Bashful",
    "Dopey", "Sleepy", "Grumpy")
```

You *must* resist the obvious temptation to format the string fully, and pass that to the logging function:

```
num_fruits = 14
fruit_name = "oranges"
logging.warning(
    "Don't do this: %d %s" % (num_fruits, fruit_name))
logging.warning(
    "Or this: {} {}".format(
    num_fruits, fruit_name))
logging.warning(
    f"And definitely not this: {num_fruits} {fruit_name}")
```

All of these work, in the sense that you will get correct log messages. But each surrenders important performance benefits `logging` normally provides. Remember: when the line of code with the log message is executed, it may not actually trigger a log

5 *https://docs.python.org/3/howto/logging-cookbook.html#use-of-alternative-formatting-styles*

event. If the log level threshold is higher than the message itself, the line does nothing. In that case, there is no reason to format the string.

In the "do it like this" form, the string is formatted if and only if a log event actually happens, so that's fine. But if you format the string yourself, it's *always* formatted. That takes up system memory and CPU cycles even if no logging takes place. If the code path with the logging call is only executed occasionally, that's not a big deal. But it impairs the program when a debug message is logged in the middle of a tight loop. When you originally code the line, you never really know where it might migrate in the future, or who will call your function in ways you never imagined. So I recommend you always use the recommended form in your logging calls.

There is another point to make about f-strings. Realize that f-strings are Python syntax for constructing string literals. When you use an f-string, the final string is rendered at the expression level. There is no way to delay that calculation or make it "lazy". This makes it impossible to use the deferred string rendering the logging functions are designed for.

So just use the supported form, where the first argument is the format string, and subsequent arguments are the parameters for it. You can also use named fields, by passing a dictionary as the second argument—though you must use the percent-formatting named-field format:

```
fruit_info = {"count": 14, "name": "oranges"}
logging.info(
    "We ate %(count)d of your %(name)s. Thanks!",
    fruit_info)
```

Beyond Basic: Loggers

The basic interface is simple and easy to set up. It works well in single-file scripts. Larger Python applications tend to have different logging needs, however. `logging` meets these needs through a richer interface, called *logger objects*—or simply, *loggers*.

Actually, you have been using a logger object all along: when you call `logging.warning()` (or the other log functions), they convey messages through what is called the *root logger*—the primary, default logger object. This is why the word "`root`" shows in some example output.

`logger.basicConfig()` operates on this root logger. You can fetch the actual root logger object by calling `logging.getLogger()`:

```
>>> logger = logging.getLogger()
>>> logger.name
'root'
```

As you can see, it knows its name is "root". Logger objects have all the same functions (methods, actually) that the logging module itself has:

```
import logging
logger = logging.getLogger()
logger.debug("Small detail. Useful for troubleshooting.")
logger.info("This is informative.")
logger.warning("This is a warning message.")
logger.error("Uh oh. Something went wrong.")
logger.critical("We have a big problem!")
```

Save this in a file and run it, and you'll see the following output:

```
This is a warning message.
Uh oh. Something went wrong.
We have a big problem!
```

This is different from what we saw with basicConfig(), which printed out this:

```
WARNING:root:This is a warning message.
ERROR:root:Uh oh. Something went wrong.
CRITICAL:root:We have a big problem!
```

This is a step backward compared to basicConfig(). The log message does not display the log level, or any other useful information. The log level threshold is hard-coded to logging.WARNING, with no way to change it. The logging output will be written to standard error and nowhere else, regardless of where you actually need it to go.

Let's take inventory of what we want to control here. We want to choose our log record format, control the log level threshold, and write messages to different streams and destinations. You do all this with a tool called a *handler*.

Log Destinations: Handlers and Streams

By default, loggers write to standard error. It is possible to select a different destination—or even *several* destinations—for each log record:

- You can write log records to a file. This is very common.
- You can write records to a file, and *also* parrot it to stderr.
- Or you can write to stdout, or both.
- You can simultaneously log messages to two different files.
- You can log (say) INFO and higher messages to one file, and ERROR and higher to another.
- You can write records to a remote log server through an API.
- You can set a different, custom log format for each destination.

This is all managed through what are called *handlers*. In Python logging, a handler's job is to take a log record, and make sure it gets recorded in the appropriate destination. That destination can be a file; a stream like stderr or stdout; or something more abstract, like inserting into a queue, or transmitting via an RPC or HTTP call.

By default, logger objects don't have any handlers. You can verify this using the `hasHandlers()` method:

```
>>> logger = logging.getLogger()
>>> logger.hasHandlers()
False
```

With no handler, a logger has the following behavior:

- Messages are written to stderr.
- Only the message is written, nothing else. There's no way to add fields or otherwise modify it.
- The log level threshold is `logging.WARNING`. There is no way to change that.

To change this behavior, your first step is to create a handler. Nearly all logger objects you ever use will have custom handlers. Let's see how to create a simple handler that writes messages to a file, called `log.txt`.

```
import logging
logger = logging.getLogger()
log_file_handler = logging.FileHandler("log.txt")
logger.addHandler(log_file_handler)
logger.debug("A little detail")
logger.warning("Boo!")
```

The `logging` module provides a class called `FileHandler`. It takes a file path argument, and will write log records into that file, one per line. When you run this code, `log.txt` will be created (if it doesn't already exist), and will contain the string "Boo!" followed by a newline. (If `log.txt` did exist already, the logged message would be *appended* to the end of the file.)

"A little detail" is not written, because it's below the default logger threshold of `WARNING`. You can change that by calling a method named `setLevel()` on the logger object:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
log_file_handler = logging.FileHandler("log.txt")
logger.addHandler(log_file_handler)
logger.debug("A little detail")
logger.warning("Boo!")
```

This writes the following in `log.txt`:

```
A little detail
Boo!
```

Confusingly, you can call `setLevel()` on a logger with no handlers, *but it has no effect*:

```
# Doing it wrong:
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG) # No effect.
logger.debug("This won't work :(")
```

To change the threshold from the default of `logging.WARNING`, you must both add a handler, *and* change the logger's level.

What if you want to log to stdout? Do that with a `StreamHandler`:

```
import logging
import sys
logger = logging.getLogger()
out_handler = logging.StreamHandler(sys.stdout)
logger.addHandler(out_handler)
logger.warning("Boo!")
```

If you save this in a file and run it, you'll get "Boo!" on standard output. Notice that `logging.StreamHandler` takes `sys.stdout` as its argument. You can create a `StreamHandler` without an argument too, in which case it will write its records to standard error:

```
import logging
logger = logging.getLogger()
# Same as StreamHandler(sys.stderr)
stderr_handler = logging.StreamHandler()
logger.addHandler(stderr_handler)
logger.warning("This goes to standard error")
```

In fact, you can pass any file-like object; The object just needs to define compatible `write` and `flush` methods. Theoretically, you could even log to a file by creating a handler like `StreamHandler(open("log.txt", "a"))`—but in that case, it's better to use a `FileHandler` so that it can manage opening and closing the file.

Most of the handlers you might need are provided for you in the `logging` module. The most common handlers you use will probably be `StreamHandler` and `FileHandler`. Others include:

- `WatchedFileHandler` and `RotatingFileHandler`, for logging to rotated log files
- `SocketHandler` and `DatagramHandler` for logging over network sockets
- `HTTPHandler` for logging over an HTTP REST interface

- `QueueHandler` and `QueueListener` for queuing log records across thread and process boundaries

See the official docs[6] for details. At times, you may need to create a custom handler, by subclassing `logging.Handler`. This lets you log to any destination you need to.

Logging to Multiple Destinations

Suppose you want your long-running application to log all messages to a file, including debug-level records. At the same time, you want warnings, errors, and criticals logged to the console. How do you do this?

We've given you part of the answer already. A single logger object can have multiple handlers: all you have to do is call `addHandler()` multiple times, passing a different handler object for each. For example, here is how you parrot all log messages to the console (via standard error) and also to a file:

```
import logging
logger = logging.getLogger()
# Remember, StreamHandler defaults to using sys.stderr
console_handler = logging.StreamHandler()
logger.addHandler(console_handler)
# Now the file handler:
logfile_handler = logging.FileHandler("log.txt")
logger.addHandler(logfile_handler)
logger.warning(
    "This goes to both the console, AND log.txt.")
```

This combines what we learned above. We create two handlers—a `StreamHandler` named `console_handler`, and a `FileHandler` named `logfile_handler`—and add both to the same logger (via `addHandler`). That's all you need to log to multiple destinations in parallel. Sure enough, if you save the above in a script and run it, you'll find the messages are both written into `log.txt`, as well as printed on the console (through standard error).

How do we make it so every record is written in the log file, but only those of `logging.WARNING` or higher get sent to the console screen? Do this by setting log level thresholds for both the logger object and the individual handlers. Both logger objects and handlers have a method called `setLevel()`, which take a log level threshold as an argument:

```
my_logger.setLevel(logging.DEBUG)
my_handler.setLevel(logging.INFO)
```

6 *https://docs.python.org/3/library/logging.handlers.html*

If you set the level for a logger, but not its handlers, the handlers inherit from the logger:

```
my_logger.setLevel(logging.ERROR)
my_logger.addHandler(my_handler)
my_logger.error("This message is emitted by my_handler.")
my_logger.debug("But this message will not.")
```

You can override that at the handler level. Here, I create two handlers. One handler inherits its threshold from the logger, while the other does its own thing:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

verbose_handler = logging.FileHandler("verbose.txt")
logger.addHandler(verbose_handler)

terse_handler = logging.FileHandler("terse.txt")
terse_handler.setLevel(logging.WARNING)
logger.addHandler(terse_handler)

logger.debug("This message appears in verbose.txt ONLY.")
logger.warning("And this message appears in both files.")
```

There's a caveat, though: a handler can only make itself *more* selective than its logger, not less. If the logger chooses a threshold of `logger.DEBUG`, its handler can choose a threshold of `logger.INFO`, or `logger.ERROR`, and so on. But if the logger defines a strict threshold—say, `logger.INFO`—an individual handler cannot choose a lower one, like `logger.DEBUG`. So something like this won't work:

```
# This doesn't quite work...
import logging
my_logger = logging.getLogger()
my_logger.setLevel(logging.INFO)
my_handler = logging.StreamHandler()
my_handler.setLevel(logging.DEBUG) # FAIL!
my_logger.addHandler(my_handler)
my_logger.debug("No one will ever see this message :(")
```

There's a subtle corollary of this. By default, a logger object's threshold is set to `logger.WARNING`. So if you don't set the logger object's log level at all, it implicitly censors all handlers:

```
import logging
my_logger = logging.getLogger()
my_handler = logging.StreamHandler()
my_handler.setLevel(logging.DEBUG) # FAIL!
my_logger.addHandler(my_handler)
# No one will see this message either.
my_logger.debug("Because anything under WARNING gets filtered.")
```

The logger object's default log level is not always permissive enough for all handlers you might want to define. So you will generally want to start by setting the logger object to the lowest threshold needed by any log-record destination, and tighten that threshold for each handler as needed.

Bringing this all together, we can now accomplish what we originally wanted—to verbosely log everything into a log file, while duplicating only the more interesting messages onto the console:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
# Warnings and higher only on the console.
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.WARNING)
logger.addHandler(console_handler)
# But allow everything to go into the log file.
logfile_handler = logging.FileHandler("log.txt")
logger.addHandler(logfile_handler)

logger.warning("This goes to both the console, AND into log.txt.")
logger.debug("While this only goes to the file.")
```

Add as many handlers as you want. Each can have different log levels. You can log to many different destinations, using the different built-in handler types mentioned above. If those don't do what you need, implement your own subclass of `logging.Handler` and use that.

Record Layout with Formatters

So far, we've only shown you how to create logger objects that will write just the log message and nothing else. At the very least, you probably want to annotate that with the log level. You may also want to insert the time and other information. How do you do that?

The answer is to use a *formatter*. A formatter converts a log record into something that is recorded in the handler's destination. That's an abstract way of saying it; more simply, a typical formatter just converts the record into a usefully formatted string that contains the actual log message, as well as the other fields you care about.

The procedure is to create a `Formatter` object, then associate it with a handler (using the latter's `setFormatter()` method). Creating a formatter is easy—it normally takes just one argument, the format string:

```
import logging
my_handler = logging.StreamHandler()
fmt = logging.Formatter("My message is: %(message)s")
my_handler.setFormatter(fmt)
my_logger = logging.getLogger()
```

```
my_logger.addHandler(my_handler)
my_logger.warning("WAKE UP!!")
```

If you run this in a script, the output will be:

```
My message is: WAKE UP!!
```

Notice the attribute for the message, `%(message)s`, included in the string. This is just a normal formatting string in the older, percent-formatting style. It's exactly equivalent to using the `format` argument when you call `basicConfig()`. For this reason, you can use the same attributes, arranged however you like (see Table 9-1).

Conclusion

The larger your program, the more valuable logging becomes. You can get along without it so long as you write nothing but small Python scripts. But having read this far, your ambitions are likely higher. For more sophisticated software systems, smart use of Python's rich logging module helps you create better software faster, and keep it running great.

Parting Words

We have come a long way, haven't we?

Congratulations on completing this book. If you have read carefully, and tried to apply what you have learned in your own coding, you now understand real-world Python at a deeper level than the vast majority of Python coders.

But in another sense, your journey has just begun. You have a great foundation of knowledge, new mental models, and powerful distinctions. The next step is to master applying them in what you do, writing Python software every day.

To help, I have created coding exercises for every chapter, plus other fun resources—exclusively for readers of this book. To get these along with email notifications of future book releases, go to *https://powerfulpython.com/register* and follow the instructions.

You can reach me by email at *aaron@powerfulpython.com*. For professional training options, go to *https://powerfulpython.com* and browse the resources there.

Creating this book has been an incredible journey. I thank you for joining me along the way.

Index

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

About the Author

Aaron Maxwell is a software engineer and Pythonista. Through a decade working in Silicon Valley engineering teams, including two Unicorns, he has gained strong production experience in backend engineering at scale; data science and machine learning; test automation infrastructure; DevOps and SRE; cloud infrastructure; marketing automation; and more. He codes in a variety of languages, including plenty of Python.

Aaron then pivoted to training, developing an innovative curriculum for intermediate and advanced Python which he has taught to over 10,000 technology professionals worldwide—in nearly every engineering domain, country, and culture.

Colophon

The animal on the cover of *Powerful Python* is a red-tailed boa (*Boa constrictor constrictor*). Also known as the Colombian boa, these large snakes can be found in the Sonoran desert in Mexico and all throughout Central America, and as far as the northern regions of Peru.

While red-tailed boas are popularly known for being the largest snake species, in reality, they are actually modest in size compared to others. Red-tailed boas are 10 to 12 feet long and can weigh anywhere between 25 and 50 pounds. They can be identified by their irregular, black oval patches that cover their tan scales; the blotches on their tails are more reddish in color. In the wild, red-tailed boas have an estimated lifespan of 15 years, but they can live up to 25 years in human care.

Red-tailed boas live in woodlands and rainforests, usually making their homes in hollowed logs or tree branches. They are nocturnal hunters and are known for ambushing their prey; red-tailed boas squeeze their prey and kill them by shutting down their vital organs and swallowing them whole. A red-tailed boa's diet usually consists of lizards, rodents, birds, eggs, and sometimes other snakes. A large meal can sustain them for up to a month.

Although the status of red-tailed boas has not yet been assessed, they do face many threats, including habitation loss and illegal pet trade. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *Dover*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.